Computer Science Department

TECHNICAL REPORT

Toward a Fully Integrated VLSI CAD System: From Custom to Fully Automatic

Yongtao You

Technical Report 522

October 1990

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

Toward a Fully Integrated VLSI CAD System: From Custom to Fully Automatic

Yongtao You

Technical Report 522

October 1990

Toward a Fully Integrated VLSI CAD System: From Custom to Fully Automatic

Yongtao You

Computer Science Department
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

October 1990

A dissertation in the Department of Computer Science submitted to the faculty of the Graduate School of Arts and Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy at New York University.

Approved: ______________________________

Professor Alan R. Siegel

Toward a Fully Integrated VLSI CAD System: From Custom to Fully Automatic

Yongtao You

Abstract

This thesis describes an integrated CAD environment, which is intented to support almost all phases of the VLSI circuit design cycle, from high-level circuit description down to mask generation. Several VLSI CAD tools have been integrated together under the environment, including a multi-level simulator, automatic placement tools, a schematic layout editor, and a UC Berkeley-developed geometry layout editor.

The multi-level simulator supports top-down design by allowing circuits whose components are described at different levels to be simulated together. The levels of circuit description currently supported include a variant of C programming language for circuit behavior descriptions, the schematic layout representation, and the Magic layout from which masks for wafer fabrication can be generated. The hardness of charge-sharing modeling problem is resolved, and a new model for it is given.

The schematic layout editor allows designers to specify interconnections among circuit components in a very efficient manner. It separates behavioral descriptions of a circuit from its geometric layout. Designers can have a graphical view of their design, and specify, within this graphical organization, the behavioral description of components at different levels of abstraction. These schematic layouts with different levels of representation can be simulated using the multi-level simulator.

The automatic placement tool presently performs bottom-up iterative improvement, with simulated annealing as its assistant when needed. Interactive graphics interface is provided which allows human intervention on intermediate as well as final layout.

Acknowledgments

Special thanks go to my thesis advisor, Prof. Alan R. Siegel, for his help, encouragement, and advise. He has been more than a thesis advisor to me.

I would also like to thank Prof. Jiawei Hong for his gracious help; and Prof. Bud Mishra for providing literatures.

It is a pleasure to thank Jianer Chen, Yue-Sheng Liu, Kaizhong Zhang, and Qun Zuo for their constant comments on this work, and for interesting discussions with them.

Finally I want to thank my wife, Wenhui Jia, for her understanding.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

During the last decade, semiconductor technology has undergone rapid evolution, which makes it possible to place hundreds of thousands of transistors on a single chip. The complexity of Very Large Scale Integrated (VLSI) circuits has outgrown human design capabilities, and Computer-Aided Design (CAD) tools have become essential for designers. In fact, CAD tools for VLSI circuit designs are also undergoing rapid evolution. Today's designers have at their disposal an entire array of automated design tools: automatic placement and routing, fault test generation, design rule checking, timing verification, various levels of simulation, silicon compilers, and a lot more. These tools greatly increase the productivity by providing assistance to circuit designers through the entire life cycle of VLSI chip design, from architectural specification to mask generation.

One aspect in CAD tool development which has been largely ignored until recently is the integration of these existing CAD tools. Each tool has its own input format, its own user interface, its own output format, and its own interpretation of input/output data. This has resulted in the situation where designers have to concentrate much of their efforts on interfacing various design tools, thereby paying less attention to the main task, circuit design. Due to the lack of integrations of design tools, many unnecessary burdens have been put on designers. These include the following.

- Managing the large amounts of data generated during the design process;
- Learning and remembering the user interaction methods for each of the tools used during the design process;
- Generating and interpreting tool specific data formats;
- Integrating a top-down design methodology with different tools at each phase of design.

Several integrated circuit design systems have been developed in the last few years, such as ChipBuster [MIB 86] and Cadlab [MGS 89]. These systems provide a common interface for various tools integrated under the system, which eliminates the need to learn different interfaces for each tool. A data base system is also provided which makes the management of large amounts of data generated during the design process a lot easier. Also, one single standard data format is used by all the tools under the integrated system so that designers do not need to transfer data from one format to another.

This thesis is another effort made towards integration of CAD tools. Its goal is to allow the designer to integrate custom and automatic design styles in a flexible and efficient manner. In particular, the system should support the top-down design methodology, which has been proven to be effective in designing large, complex systems. Designers should be able to design circuits at very high level of abstraction such as behavior descriptions in some hardware description languages, as well as at very low level of abstractions such as geometry layout. Also, the transformation between these levels should be easy and reliable.

We have planned to integrate as many CAD tools under the environment as necessary in order to reach this goal. The system was designed as a platform capable of supporting high performance subsystems built as needed. The multi-level simulator and the automatic placement tool were selected as our starting point. A schematic layout editor was added later. The VLSI painting and circuit extraction system Magic (designed at UC Berkeley) and various tools associated with it, such as Crystal and SPICE, was also integrated under the environment. The system is implemented on Sun Workstation under SUN/UNIX operating system.

In the rest of the thesis, we are going to explore two important problems of VLSI CAD tool family, namely simulation and placement. The main reason for choosing these two topics is that they are two of the most critical utilities needed by designers, and are suitable for an individual effort in CAD construction. Some attentions will also be given to the schematic layout editor.

1.1 Simulation

Various levels of simulators have been one of the most important members of VLSI CAD tool family. Although the use of these simulators can not guarantee the correctness of a design, they can, by using a carefully selected set of tests, identify many of the most frequently occurring types of error at earliest possible stage. Up to now, many simulators have been developed, each simulates circuits at a different level of abstraction, such as system level, functional level, gate level, switch level, and circuit level. Although separate simulators are available for almost every stage of a chip design, this multiple simulator approach has several problems:

- The design effort is increased due to the need to learn several simulation tools.
- As a result of top-down design methodology, most of the time during the design process, different components of the circuit are described at different levels of abstraction. Since each simulator operates at just one level of abstraction, only part of the circuit can be simulated at one time. Interconnections between parts that are described at

different levels can not be simulated.

- The effort needed to manually generate test data for a portion of the circuit is usually excessive. Interpreting and analyzing test results of a portion of the circuit can also be time consuming.

- Simulating the whole circuit repeatedly at a low level each time a small change is made is both unnecessary and time consuming.

The main drawback here is that the top-down design methodology is not supported. The whole circuit has to be described at the same level of abstraction in order to be simulated as a whole. The recognition of these problems leads to the development of a *multi-level simulator*, which can simulate circuit with different components described at vastly different levels of abstraction. It allows designers to start with a very high level specification of a circuit, and then refine the design one small piece at a time. After any refinement, the whole circuit with the interconnected modules[1] can be simulated even though components may be specified at any level. By verifying the design after each refinement, design errors can be found and corrected as early as possible.

As part of our integrated CAD system, a multi-level simulator has been developed. It supports the top-down design methodology by accepting any hierarchical mix of modules designed in Magic, in schematic layout, and in CHDL (a superset of C programming language). We will discuss it in detail in the next Chapter.

1.2 Placement

Generally speaking, the problem of *placement* encountered during the physical design of VLSI chips is to determine locations for its components. One closely related problem is

[1]We use *module, circuit component*, and *cell* interchangeablely throughout this thesis.

routing, which determines how these components are to be interconnected. Together these two processes determine the final layout of the circuit.

For different design styles, the placement problem is slightly different. Consider, for example, three of the most standard styles of chip design: gate-array, standard-cell, and custom. In *gate-array* design, components are low level functional blocks of rectangular shape with various sizes. The circuit is partitioned, by grid, into small rectangles called *slots*, and the placement process involves assigning components into slot or multiple slots so as to optimize any number of objective functions, such as minimizing the total interconnection length and the number of utilized components. That is, each component occupies one or more slots. In *standard-cell* design, components are of rectangle shapes, often with the same height but different width. The circuit layout is partitioned into rows with about the same height as components, and the placement process involves arranging the components so that they abut in rows, and a number of objective functions, such as total circuit area, are optimized. Component placement in *custom* design is the most complex task among the three design styles. The component are of arbitrary shape and size, and they can be placed anywhere within the layout. To cope with the complexity, most automatic placement tools allow components to have only a small set of regular shapes (e.g. rectangles, L-shapes, T-shapes, etc.), although usually no dimension restrictions apply.

Many automatic placement tools have been developed to assist designers in physical design of circuits. All of them use heuristics, which means that their performances are good on some circuits and poor on others. This is due to the fact that even the simplest placement problems in any of the three cases are of NP-complete for very reasonable objective functions such as minimizing the total area or total wire length [Don 80], [SaB 80].

As part of our integrated CAD system, we have developed an experimental placement tool. Designers can experiment and modify layout with different objective functions and search procedures. They can even place part of the circuit by hand, which may greatly improve the design for some hard-to-place circuits.

We will present in more detail the placement problem in general in Chapter 4, and the experimental automatic placement tool implemented in our integrated CAD system in Chapter 5.

Chapter 2

Principles of Multi-Level Simulation

To cope with the rapid increase of complexity of VLSI design task, various kinds of simulators have been developed. Although these simulators have been very helpful in aiding designers to build complex VLSI circuit, the needs for multi-level simulator have become more and more evident. The idea behind multi-level simulation is to build a simulator that accepts circuits with different components described in different levels of abstraction. This kind of description is a natural result of any top-down design methodology.

A multi-level simulator supports the top-down design methodology by allowing designers to start with a specification of what he/she wants to build in a very high level of abstraction. The whole circuit is usually decomposed into a number of components, and the functional behaviors of these components as well as their interconnections are then described. At this stage, the circuit should be simulated using high level features of a multi-level simulator, since this is the best time to correct any errors.

As the design process continues, components are decomposed into smaller and smaller components, and the levels of abstraction in which these components are described becomes lower and lower. This implies a hierarchical tree structure[1] in which each internal node

[1]The structure could be a DAG as the result of common subexpression eliminations.

represents a supercomponent; each leaf may represent a simple component that could be implemented directly at the lowest level, or it might represent a supercomponent that will be decomposed further. Each refinement step consists of decomposing one of the supercomponents, usually a small part of the circuit, into smaller components, and producing a description for each newly created components at some level of abstraction (which is usually at a lower level than that of the parent). After each refinement step, it is a good idea to simulate the whole circuit in order to find possible errors made during the refinement step. This suggests the need of multi-level simulator.

2.1 Levels of Simulation

In order for a circuit to be simulated by computer, it should first be represented by a mathematical model. Various models are used for this purpose; each corresponds to a different level of abstraction of the circuit. Of course, each model has advantages and disadvantages. Some models allow designers to specify more accurate and detailed information of the circuit than other models do, and we say these models are of lower level relative to the other models. Usually, simulators operating at lower levels of models perform more accurate simulation than those operating at higher levels. Every part of the circuit should be simulated at very lowest possible level sometime. But they usually consume much more time. On the other hand, lower level descriptions of the circuit is not available at early stages of design, so high level simulators are necessary. This explains why simulators at various levels exist at the same time.

In the following subsections, we are going to discuss five models that are used most often, and their advantages and disadvantages.

2.1.1 Architectural-Level Simulation

Simulations of this kind are employeed at the system design stage to predict performance of the system, and to determine architectural parameters of the system, such as the amount of cache, or the number of registers. Behaviors of components are usually specified in a very incomplete way. Only those that are relevant to system performance are specified, and only relevant parts are simulated. This level of simulation is very different from the other levels of simulation that we are going to discuss below, both in concept and in implementation.

Architectural-level simulations are usually done in an *ad hoc* way because of the lack of good mathematical models [OrR 83], [SSS 87].

2.1.2 Functional-Level Simulation

At this stage of design, systems have been divided into smaller components. The functional behaviors of these components and the interactions between them are specified. Usually, no structural information is given at this time. For the purpose of functional behavior description, as well as others, many specification languages, often called Hardware Description Languages (HDL), are invented. While some of these HDLs were created from scratch, like VHDL [LMS 86], some were simply modifications of existing programming languages, like ADLIB [HiC 87] from Pascal, and SIMMER [LaK 85] from LISP. For simplicity, we chose a superset of C programming language as our HDL. Simulations at this level are used to find behavioral specification faults as soon as possible; these faults are not worthy of the significant amount of effort required to instantiate them in a lower level implementation.

Functional-level simulators are usually very fast, since they ignore lots of details of the circuit, such as structural information and how the circuit will be implemented at lower level. They usually act like a abstract mapping between inputs and outputs. Most programming techniques can be used to speed up the simulation. Furthermore, specifications at this level are usually treated as the definitions of the circuit to be designed. The correctness of designs at lower levels is checked with the specifications of the functional level.

2.1.3 Gate-Level Simulation

At this level of abstraction, the primary building blocks of circuits are gates, e.g. AND gate, OR gate, etc. connected by memoryless nets. Usually, only a limited number of predefined building blocks are available to the designer. Circuits are described in some kind of HDLs, and the specifications are usually produced directly by hand.

Gate-level simulations and functional-level simulations share lots of common aspects. They are very similar in the sense that both gates and functional blocks act as mappings between their inputs and outputs, although gates are usually simpler and functional blocks are usually more complex; they both have memoryless nets as their interconnections; and signal flows between them are logically unidirectional. Perhaps the only difference is the complexities of their components. So it is much easier to mix these two levels together than with other levels of simulation. Maybe that is why for many multi-level simulators, this is the lowest level of description they could handle.

2.1.4 Switch-Level Simulation

While the gate-level model is a good low level representation for circuits built in TTL or ECL technology, it is one level too high for circuits built in MOS technology. In MOS circuits, the basic building blocks are various types of transistors connected by wires capable of "remembering" the previous state for a short period of time. Signals can flow bidirectionally. And circuits can be implemented directly from transistors, not just from those limited number of predefined gates.

Since the introduction of switch-level model by Bryant [Bry 84], many switch-level simulators have been developed [Bry 87]. On the one hand, they can capture phenomenon such as bidirectional signal flow, charge sharing, etc. , thus providing more accurate simulation than gate-level simulators. On the other hand, by using a discrate model instead of a linear model, and by efficient implementation, most of the switch-level simulators achieve the speed approaching that of the gate-level simulators.

2.1.5 Circuit-Level Simulation

Perhaps the the most time consuming as well as the most accurate simulator currently available would be a circuit-level simulator, such as SPICE [Nag 75] and RELAX [LeS 82]. It goes into the detailed electrical behavior analysis of the design by solving a set of non-linear equations. "Estimates show that circuit simulation of a single multiply instruction using SPICE on a 450,000 transistor CPU chip would take approximately 6 CPU months to complete on an IBM 370/168 processor, and require 250Mb of memory." [ONC 86] Due to the enormous amount of time consumed, large circuits are seldomly simulated at this level. Instead, only a very small portion of the circuit is simulated at one time. Designers first isolate the possible faults into a very small region, with the assistance of other tools such as a timing simulator or a switch-level simulator. This small region is then simulated at the circuit level, using specially derived input data.

2.2 Multi-Level Simulation

The problem with conventional multi-simulators approach is that no one simulator is useful throughout all phases of design. Usually, several types of simulators are used at various stages of design. Since each simulator is capable of simulating at one level of abstraction, the whole design has to be described at that same level in order to be simulated by the simulator. If only part of the design is described at this level, you can also simulate this part, but only this part. The interactions of this part with the rest of the system, for instance, can not be simulated. This limits the use of top-down design methodology, which has been proven to be effective in designing large systems.

Given these problems and limitations, many attempts have been made towards the direction of multi-level simulation, such as Multi-Sim [ChC 78], SALOGS-IV [CaS 78], the mixed-mode simulator in [ABK 80], and SABLE/HELIX [HiC 87]. For example, the S-ABLE/HELIX provides users with a multi-level hardware behavioral description language ADLIB based on Pascal, and a structural description language SDL. While the ADLIB is used to describe behavioral information of a circuit, the SDL is used to describe structural information. Both languages are multi-level in the sense that ADLIB provides users with different data structures and their interpretations, and SDL allows users to describe structure of a circuit as a hierarchy of components. Its advantages as a multi-level simulator are due mainly to the multi-level description language ADLIB. For example, a memory address could be represented as an integer (at a higher level), or it could be represented as an array of real numbers denoting the instantaneous voltages on a set of address lines (at a lower level). Circuit components described at different levels could be simulated simultaneously.

Although all of these systems share some of the merits of a multi-level simulator, each has its own drawbacks.

- Some of the multi-level simulators was designed only for a special class of circuits. For instance, Multi-Sim was designed solely for microprocessor based systems.

- Although allowing modules to be described in multi-levels of abstraction, some simulators, like Multi-Sim and the mixed-mode simulator described in [ABK 80], have only a limited number of predefined modules available at functional level. Users can not define his/her own modules at the functional level.

- For low level circuit descriptions, some simulators, as of Multi-Sim, SALOGS-IV, ADLIB simulator, and the mixed-mode simulator in [ABK 80], only allow users to design his/her circuit in some kind of description languages, instead of extracting circuit descriptions from layout. Although useful, this kind of description cannot be used as a substitution of layout extractions, since such a design is only what designers think the circuit would be, not what will be fabricated. Furthermore, a HDL design would require every transistor and connection to have a textual specification as well as a graphical description, and the consistency between the two designs would be based upon human vigilance.

- For some multi-level simulators, gate-level simulation is the lowest level of simulation that they can perform, while VLSI circuit needs switch-level simulation.

Chapter 3

MSIM: The Multi-Level Simulator

The goal of our multi-level simulator is to allow different part of a circuit to be described in different HDLs, at different levels of abstraction, and still be able to be simulated together. Much effort has been taken during implementation so as to permit new HDLs to be added to the multi-level simulator easily. The simulator is an interactive, event-driven multi-level simulator which operates in unit delay mode. So far, two levels of the circuit description are recognized by our multi-level simulator, the functional-level and the switch-level.

Some of the features of our multi-level simulator are:

- Acceptance of different levels of VLSI circuit descriptions;
- Linear time race detection;
- Linear time switch-level simulation algorithm;
- Unlimited number of levels of node size and transistor size;
- Choice of simulating at high levels of abstraction for speed, or low levels of abstraction for accuracy;
- Compatibility with Magic.

3.1 Switch-Level Simulation

Our switch-level simulation is based on the switch-level model introduced by Bryant [Bry 84]. It accepts as input the net list extracted from geometrical layout produced by Magic (an U.C. Berkeley designed VLSI painting and circuit extraction system). Although using the same mathematical model as MOSSIM does, our simulator has a completely different implementation. Instead of solving linear equations, we do a graph traversal using the *best-first search* technique. Also, entensive table lookups are adopted to speed up the simulation. Another feature implemented is a linear time race detection algorithm.

3.1.1 The Network Model

In this section, we present the network model introduced by Bryant (See [Bry 84] for details.) for switch-level simulators.

In Bryant's switch-level model, a MOS circuit is modeled as a set of nodes $\{n_1, n_2, \ldots, n_m\}$ connected by a set of transistors $\{t_1, t_2, \ldots, t_n\}$. Each node n_i has a state in the set $T = \{0, 1, X\}$, and is classified as either an *input node* or a *storage node.* For each storage node, there is a size in the set $\{\kappa_1, \kappa_2, \ldots, \kappa_m\}$ associated with it. The size of a node indicates its approximate capacitance relative to other nodes with witch it may share charge, where sizes are ordered $\kappa_1 < \kappa_2 < \cdots < \kappa_m$. A node with state of 0 means the presents of a low voltage signal; a node with state of 1 means the presents of a high voltage signal; and a node with state of X means the presents of a unknow voltage signal. No attempt is made to distinguish between "unknown but valid" state (a state in which node has a valid voltage but it's not known to the simulator) and "invalid" state (a state in which node has a invalid, intermediate voltage). Input nodes are sources of electrical current, like Vdd and GND. They have size ω to distinguish them from storage nodes.

The basic building block in MOS circuit is the *transistor*, which is used primarily as a switching element. A transistor is a three terminal device with terminals labeled "gate", "source", and "drain". The source and drain terminals are symmetric; signals could be transmitted bidirectionally between them, controlled by the voltage at the gate terminal. Each transistor t_i also has a state in the set $T = \{0, 1, X\}$, but with 0 indicating an open (disconducting) switch, and 1 indicating a closed (conducting) switch. A transistor in the X state forms an indeterminate conductance between (inclusively) its conductance when open and that when closed. Also, each transistor has a strength in the set $\{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ indicating its conductance when closed relative to other transistors which may form part of a ratio path, where strengths are ordered $\gamma_1 < \gamma_2 < \cdots < \gamma_n$.

There are three types of transistors, each behaves differently in response to the voltage at the gate, as shown in Table 3.1.

gate state	**n**-type	**p**-type	**d**-type
0	open	closed	closed
1	closed	open	closed
X	unknown	unknown	closed

Table 3.1: Transistor types

For instance, for an **n**-type transistor, a high voltage at its gate causes a high conductance path between the source and the drain; and a low voltage at the gate isolates the source from the drain. A **d**-type transistor corresponds to a depletion mode transistor, which is always closed.

3.1.2 Algorithm

The algorithm we used was based on the network model described in the previous section. Under the so called *unit delay* mode, every transistor takes the same amount of time to switch. And signals take the same amount of time to go through any wires, no matter how long or how short it is. So a combinational network n levels deep takes exactly n time units to stabilize. Within each time unit, all the transistors whose gates have changed their states will switch first, and then all the signals will travel through "closed" (conducting) transistors as far as they can go.

Formally, the final state of the circuit after one unit step is determined by the so called *steady-state response* function. If no X state is present in the circuit, the steady-state response function $F(\boldsymbol{y}, \boldsymbol{z})$ equals the vector of node states $\boldsymbol{y}'$ that would result if the nodes were initialized to states given by the vector $\boldsymbol{y}$, and the transistors were held fixed in states given by the vector $\boldsymbol{z}$, where $\boldsymbol{y} \in \{0,1\}^m$ and $\boldsymbol{z} \in \{0,1\}^n$. For the case where $\boldsymbol{y}, \boldsymbol{z} \in \{0,1,X\}^m$, we want to define the steady-state response on node n_i as 0 (or 1) iff it would have this same steady-state response if the nodes initially in the X state were set to any combination of 0's and 1's and the transistors in the X state were set to any combination of 0's (open) and 1's (closed). To compute the steady-state response, we need the following definitions [Bry 84].

Definition 3.1 *The* **least upper bound** *(lub) of a set of ternary values equals 1 (or 0) iff all elements of the set equal 1 (or 0), and equals X otherwise. The lub operation acts as a "consistency" operation with inconsistency represented by X.*

Definition 3.2 *The* **ternary switch graph** *S of a circuit contains a vertex v_i for each node n_i in the circuit, with size $Size(v_i)$ equal to the size of node n_i, and with state $State(v_i)$ equal to the initial state of node n_i. S contains a 1-edge for each transistor in the circuit whose initial state is* 1*; and an X-edge for each transistor whose initial state is X. Edge e_i (either 1-edge or X-edge) connects the vertices corresponding to the source and drain of transistor t_i and has* strength *$Strength(e_i)$ equal to the strength of this transistor.*

The effect of the initial voltage of one node on the steady-state voltage of another through a series of conducting (maybe X) transistors is described in terms of a *rooted path.*

Definition 3.3 *A rooted path p in a ternary switch graph is a quadruple ⟨ Root(p), Dest(p), 1-Edges(p), X-Edges(p) ⟩ consisting of an initial vertex Root(p), a final vertex Dest(p), a set of 1-edges 1-Edges(p), and a set of X-edges X-Edges(p), such that the elements of the set Edges(p) = 1-Edges(p) ∪ X-Edges(p) form a contiguous simple path from Root(p) to Dest(p). The strength of p, denoted |p|, is equal to*

$$|p| = min(Size(Root(p)),\ Strength(e_i) \mid e_i \in Edges(p)).$$

Definition 3.4 *A rooted path p is called a* **definite** *path if X-Edges(p) = ∅.*

Definition 3.5 *A rooted path p in a ternary switch graph is said to be* **blocked** *iff for some initial segment p′ of p and some definite rooted path q, Dest(p′) = Dest(q) and |p′| < |q|.*

It is easily seen that if there is an *unblocked* path from node n_j to node n_i, that means the signal from node n_j is one of the strongest among all that can reach n_i, hence the current state of n_j will affect the final state of n_i. Let's define $jP(z)i$ iff there exits an *unblocked* rooted path p in a ternary switch graph such that $Root(p) = v_j$ and $Dest(p) = v_i$. Then the steady-state response of node n_i for initial node state $\boldsymbol{y}$ and transistor state $\boldsymbol{z}$ is given by $\boldsymbol{y}'$:

$$y_i' = lub \ \{ \ y_j \mid jP(z)i \ \}. \tag{3.1}$$

From the above definitions we can see that the steady-state response function can be computed by traversing the switch graph three times. During the first traversal, we compute all the definite paths. During the second traversal, we try to find all the unblocked paths starting at a node with initial state of 1 or X; and during the third traversal, find all the unblocked paths starting at a node with initial state of 0 or X. Then use Equation 3.1 to compute the steady-state response of each node. If no unblocked paths from a node with initial state of 1 of X to node n_i has been found, then the steady-state response for n_i will be 0; if no unblocked path from a node with initial state of 0 or X to n_i has been found, then the steady-state response for n_i will be 1; otherwise it will be X.

Algorithm 3.1 *Switch-level Simulation Algorithm.*

INPUT: *A list of nodes and a list of transistors representing a circuit. Each node in the list has an initial state, which may be the result of previous simulations, or X. Each transistor in the list also has an initial state. Also provided as input is an event queue of nodes with their new states.*

OUTPUT: *The output of this algorithm is reflected by the side effects on the list of nodes given as input. As a result, some nodes will be set to a new state.*

METHOD:

1. *If the event queue is empty, or the time limit exceeds, return.*
2. *For each node on the event queue, find those transistors of which it is the gate and change their states.*
3. *For each node on the event queue, find those transistors of which it is the gate, and put the transistor's source and drain, and all the unqueued nodes reachable from them onto a multi-level queue.*
4. *Push signals on the multi-level queue through "on" transistors, strongest first. Each time a signal reaches a node, remember the signal with its strength at the node. After this step, all definite paths are found.*
5. *Push signals on the multi-level queue through "on" or "X" transistors, strongest first. Each time a signal reaches a node, remember the signal with its strength at the node. Note that the definite paths found in step 4 are checked to see if any signals being pushed are blocked.*

6. *Empty the event queue.*

7. *For each node on the list, compute its new state using Equation 3.1. If the new state is different from the old one, put the node onto the event queue.*

8. *Goto step 1.*

□

3.1.3 Implementation

The main part of Algorithm 3.1 is to find the strongest paths of certain kind for each node. As we can see from Algorithm 3.1, at least two passes through the ternary switch graph are needed before we can use Equation 3.1 to compute the steady-state response for each node. First we have to find all definite paths, by traversing the ternary switch graph once, so that we know which paths will be blocked during subsequent passes. Next all signals are pushed through "on" or "X" transistors, as long as they are not blocked by any definite paths found in the previous pass. Since a definite path can block a signal only if it is stronger than the signal, by pushing signals in a right order, we can traverse the ternary switch graph only once. We may encode all signals into one record and push them together, as long as the definite paths are in place before we push other signals that they should block. Our way of doing this is by *best-first search*, a variant of breath-first search, where instead of choosing the leftmost child to expand, we choose the "best" one (the one with the strongest signal) to expand. To do so, we keep $m + n$ groups of sublists, one for each strength in the set $S = \{\gamma_n, \ldots, \gamma_1, \kappa_m, \ldots \kappa_1\}$. The group in which a node belongs is determined by the strength of the *definite* signal[1] on the node. Within each group, there are at most $m + n$ sublists, and the sublist on which a node belongs is determined by the strongest strength of the unblocked, non-definite signal on the node. These sublists are used in the usual way as

[1]There is a definite signal on a node n iff there is an unblocked definite path p such that $Dest(p) = n$. The strength of this definite signal equals $|p|$.

an event list is used in most event driven simulators, except that we split them into several sublists. The purpose of dividing the event queue into sublists is to simplify the selection of the "best" during the best-first search. Signals in group γ_n are propagated first, then those in group γ_{n-1} are propagated, and so on. Within each group, signals on the sublist corresponding to the strongest strength are propagated first. Here we are giving priorities to definite signals since they are capable of blocking other signals. What we want to do is to push the strongest signal first, so some of the weaker signals will be blocked when we try to push them further, and they will just be ignored. This way, we can find the strongest path for all nodes in time $O(s+t)$, where s is the size of the set S, and t is the number of transistors in the circuit.

3.1.4 Race Condition Detection

A node in a circuit is said to have the potential of having *race condition* if its final state depends on the relative speeds of signals reaching it. In another word, a race condition occurs at a node if it settles at different states for different orders of switchings of the gates in the circuit. Fig. 3.1 gives an example of a circuit in which a race condition occurs.

In Fig. 3.1, after phase $\overline{phi}$, both A and B will settle to 1. During phase phi, the A node is switching from 1 to 0, as the phi is switching from 0 to 1. Depending on how fast the two transistors s_1 and s_2 switch relatively to each other, the node B could end up in 1 or 0. If the transistor s_2 switches faster than the other, then B will remain in 1 as been set during phase $\overline{phi}$, at least for a while. On the other hand, if the transistor s_1 switches faster, then there will be a path from B to the ground momentarily. If this path could be kept long enough, B could be driven to 0. So we say a race condition occurs at node B.

A switch can be turned on in our switch-level simulator to detect race conditions. The technique used here to detect race conditions is called *ternary simulation* [Bry 83]. It works

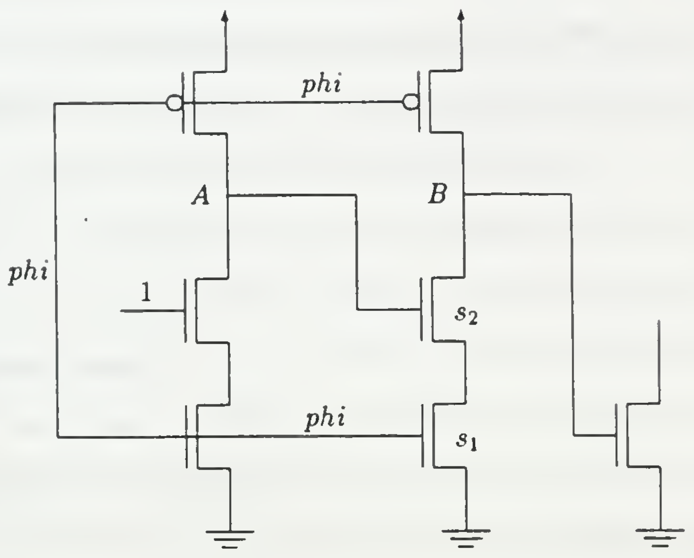


Figure 3.1: Race condition at node B

by inserting another phase, called *transition phase*, between each phases at which input data or clocks are changed. At the beginning of each phase, just as we are going to change inputs or clocks, we first set the changing inputs to X and the network is simulated just as usual until a stable state is reached. As a consequence, all nodes which could possibly change state during this phase are set to X. This is because a node could change its state only if some of the switching transistors could cause different values to reach it. And by setting those transistors to X still allows the same values to reach the node, except that they are not coming along a *definite path*, hence setting the node to X state. Next, all the changing inputs are set to their final values and the network is again simulated until a stable state is reached. As a result, any node for which the final logic level depends on the particular logic or wiring delays in the circuit will remain set to X, indicating a race condition or even a possible sequential timing error. The reason is that a node which got an X during the transition phase means that it is possible for it to change its state, and its state remains in X after the following phase means that there exists no definite path that brings a stronger signal to override the X. Furthermore, a kind of hazard on a node is indicated by a state sequence of the form $0 \rightarrow X \rightarrow 0$ or $1 \rightarrow X \rightarrow 1$.

Algorithm 3.2 *Race Detection Algorithm.*

INPUT: *A list of nodes and a list of transistors representing a circuit; and an event queue Q of nodes with their new states.*

OUTPUT: *The output of this algorithm is reflected by the side effects on the list of nodes given as input. Some nodes will be set to a new state.*

METHOD:

1. *Construct a new event queue Q' consisting of the same nodes as on Q. Set the new state of each node on Q' to X.*
2. *Invoke Algorithm 3.1 to simulate the circuit, using the new event queue Q'.*
3. *Again, invoke Algorithm 3.1 to simulate the circuit, but this time use the old event queue Q.*
4. *For each node n_i on the node list, let s_i^0 be its original state; let s_i^1 be the state after step 2; and let s_i^2 be the state after step 3. If $s_i^2 = X$ then report race condition at node n_i; If $(s_i^0, s_i^1, s_i^2) = (0,X,0)$ or $(s_i^0, s_i^1, s_i^2) = (1,X,1)$ then report hazard at node n_i.*

□

In our example in Fig. 3.1, at the beginning of phase *phi*, we will set *phi* to X and simulate the circuit. Obviously, both A and B will settle in X after this simulation. Then we will set *phi* to its final state, 1, and the circuit is simulated again. First, s_1 will be turned on while s_2 is in unknown state. This will not change the state of B. Then s_2 will be turned off, which is still not going to change the state of B. So B will remain in X state after phase *phi*, indicating a race condition.

3.1.5 The Simulation of Charge Sharing

The situation of *charge sharing* occurs when a node in a circuit is connected electronically to nodes storing opposite values. Fig. 3.2 gives one such example.

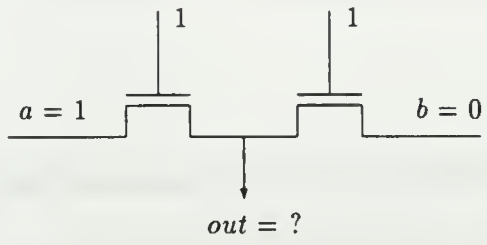


Figure 3.2: Charge sharing

In reality, the node *out* might get state of 1 if node a has more charges than b does; it might get state of 0 if node b has more charges than a does; or it might go to an undefined intermediate state if both node a and b have about same amount of charges. In other words, the signal carried by the higher capacitive line or the line with the stronger current source will establish itself at the node. Usually this is considered to be a risky practice and should be avoid. In some cases, though, it is very useful to take advantage of charge sharing, as in precharged circuit. For example, a bus, which has a huge capacity and therefore requires lots of time to be fully charged, is sometimes precharged during one clock phase and shares its charges with other nodes during the other clock phase in a two-phase clocking scheme. The reason this organization can be useful is that MOS transistors can discharge a bus more quickly than they can charge it: there is a lack of symmetry in switching capabilities.

This charge sharing phenomenon created a new problem for our switch-level simulator, because our assumption that a stronger signal could override any number of weaker signals is not so accurate in this case. In the situation of charge sharing, several weaker signals might override a stronger signal with opposite state, as long as the sum of the charges on weaker signals are significantly bigger than charges on the stronger signal. So how do we calculate steady-state response for nodes sharing charges? A natural thing to do, as Baker and Terman did (for circuit with no X-transistors) [BaT 80], is to go through the connected nodes and sum up the capacitances for each logic level. Then, depending on the ratio of the total capacitance for one level to sum of the others, compared with a predefined threshold, nodes will be assigned a final state. In more precise terms, it can be described as follows. Let ${s^1}_i$ be the sum of strengths of 1 signals that can reach node n_i; let ${s^0}_i$ be the sum of strengths of 0 signals that can reach node n_i; and let ${s^x}_i$ be the sum of strengths of X signals that can reach node n_i. To determine the final state of a node n_i in a circuit with no X-transistors, accumulate all signals reaching n_i; and assign 1 to n_i if the ratio ${s^1}_i/({s^0}_i + {s^1}_i + {s^x}_i)$ is greater than the predefined threshold; assign 0 to the node if the ratio ${s^0}_i/({s^0}_i + {s^1}_i + {s^x}_i)$ is greater than the predefined threshold; and assign X to the node otherwise. But, with the presentation of X-transistors, the problem becomes harder. Since we cannot try all possible mappings of X-transistors to $\{0,1\}$, one conservative and reasonable way of doing this is to let a node to be assigned 1 only if under no possible mappings of X-transistors to $\{0,1\}$ will result in a 0 or X for that node. Similarly, we let a node to get 0 only if under no possible mappings of X-transistors to $\{0,1\}$ will result in a 1 or X. Unfortunately, this problem is a nondeterministic polynomial (NP) complete problem, meaning that we have to try all possible mappings of X-transistors to $\{0,1\}$ to find out if one of them allows a node to reach a state. This explains why researchers, as in [BaT 80], [Bry 84], go back to the general assumption that a single stronger signal override any number of weaker signals, when charge sharing are combined with X-transistors.

Now we prove the NP-completeness of implementing the linear capacitance model for charge sharing, accompanied with the existence of X-transistors.

The *charge sharing problem* we have here is: Given a circuit, for a particular node n_i, find a mapping of X-transistors to $\{0, 1\}$ such that the ratio ${s^1}_i/({s^0}_i + {s^1}_i + {s^x}_i)$ (or ${s^0}_i/({s^0}_i + {s^1}_i + {s^x}_i)$) is maximized. If this ratio is less than the threshold, then we know that no mapping will allow node n_i to get 1 (0). A simplified version will be that ${s^x}_i = 0$. It is easy to see that the simplified charge sharing problem is equivalent to the following graph problem.

Definition 3.6 *Given an undirected graph $G = \langle V, E\rangle$, with vertices divided into two types: that of "black" vertices (B) and that of "white" vertices (W). That is, $V = B \cup W$, and $B \cap W = \emptyset$. Associated with each $v \in V$, there is a weight $w(v)$ (positive real number). For a specific $v_0 \in V$, the "b/w" ratio problem for this vertex is to find a connected subgraph $G' = \langle V', E'\rangle$ such that $v_0 \in V'$, and the "b/w ratio"*

$$\frac{\sum_{v \in B \cap V'} w(v)}{\sum_{v \in V'} w(v)} \tag{3.2}$$

is maximum. Similarly, for $v_0 \in V$, the "w/b" ratio problem is to find a connected subgraph $G' = \langle V', E'\rangle$ such that $v_0 \in V'$, and the "w/b ratio"

$$\frac{\sum_{v \in W \cap V'} w(v)}{\sum_{v \in V'} w(v)} \tag{3.3}$$

is maximum.

It is easy to see that for a "black" ("white") vertex v, the obvious solution to the "b/w" ("w/b") ratio problem is the subgraph containing a single node v and having no edges. But it can be shown that the "b/w" ("w/b") ratio problem for a "white" ("black") node is NP-complete. To prove it, we need the following definition.

Definition 3.7 *Let $Z_0{}^+$ denotes the set of positive integers, including zero. Given an undirected graph $G = \langle V, E \rangle$, a weight $w(e) \in Z_0{}^+$ for each $e \in E$, and a subset $R \subseteq V$. The* **steiner tree** *relative to R is a subtree of G that includes all the vertices of R and such that the sum of the weights of the edges in the subtree is minimized.*

It has been proved that the problem of find a steiner tree is NP-complete, even if all edge weights are equal [GaJ 79].

Theorem 3.1 *The "b/w" ratio problem for a "white" vertex is NP-complete. Similarly, the "w/b" ratio problem for a "black" vertex is NP-complete.*

Proof: Let's prove that the "b/w" ratio problem for a "white" vertex is NP-complete. The "w/b" ratio problem for a "black" vertex can be proved similarly. We will reduce the steiner tree problem [GaJ 79] to our "b/w" ratio problem. Assume we could solve the "b/w" ratio problem in polynomial time. We want to show that we could find a steiner tree in polynomial time. Let $G = \langle V, E \rangle$ be the given graph in which we want to find a steiner tree, relative to R, where $|V| = n$. Let M be the maximum of those edge weights. Choose one of the vertices in R to be v_0, and assign a weight nW to it, where $W > n^2 M$. Make v_0 a "white" vertex, make the rest of the vertices in the graph be "black". Assign all $v \in R$ the same weight W. Assign all $v \in V - R$ the same weight 0. For each $e \in E$, insert a "white" vertex with weight equal to the weight of the edge e. Clearly, the weight of any "white" vertices is less than W/n^2. Then finding a steiner tree in the original graph is equivalent to finding the maximum "b/w ratio" in the modified graph. All vertices in R must be included into the tree since all other "white" vertices are simply too small to matter. Even at the cost of including all "white" vertices, adding a single vertex in R can be beneficial. On the other hand, we want to include as few "white" vertices as possible.

$\square$

To avoid the difficulty of solving the charge sharing problem, we have to accept the less time-consuming as well as less accurate approach, which is the one we used for general cases. We again assume that a stronger signal will override any number of weaker signals with opposite state. We would like to point out though that this model is accurate enough in practice, since in a well designed circuit, the situation in which many nodes share charges does not occur very often. If we choose the threshold to be low enough, a stronger signal will actually override several weaker signals. Another advantage of this model is that the steady-state response under this model can be calculated in a uniform way as in the other cases. This makes the time complexity of our entire simulator to be linear to the number of nodes plus the number of transistors.

3.2 Functional-Level Simulation

In our implementation, the language we choose for the functional-level circuit description is a superset of C programming language, called CHDL. The reason for choosing it is that it is easy to implement. Being written in C, our switch-level simulator is easy to link with other C programs. But the simulator has been designed so that other hardware description languages, such as VHDL, could be added later without much difficulty.

3.2.1 Definition of CHDL

First of all, CHDL is a superset of C programming language. All the powers of C can be used in describing behaviors of a circuit. Beyond that, we have introduced the following keywords.

bit declares variables to be nodes. A variable of type **bit** could have a value of either 0, 1, or X.

byte declares a variable to be a list of 8 nodes. A variable **x** of type **byte** could be equivalently declared as `bit x[8];`

cell declares the identifier following it to be the name of the current cell.

in declares variables to be input ports of the current cell.

map indicates the start of a new statement, called map statement. The format of a map statement will be described later.

out declares variables to be output ports of the current cell.

simulate tells the simulator to simulate the cell at the level specified by the following identifier, such as C or MAGIC.

state declares variables to be nodes that have sufficient amount of capacities to remember its state across function calls during simulation.

use declares a cell or an array of cells as subcells of the current cell. Only cells so declared could be used in the **map** statement.

Several copies of the same subcell could appear simultaneously in a cell. They could be organized as an one or two dimensional array if appropriate. This will allow us to access them through index, and hence taking advantages of C loop statement. If these copies are unrelated, though, they can be included separately. Each copy will be assigned an *occurrence number* to distinguish from one another.

The format of a map statement is as follows:

```
map subcell-name(input-pair1, input-pair2, ...; output-pair1, output-pair2, ...);
```

In the above map statement, the `subcell-name` indicates an instance of a subcell, and has the form of either `name[occur]`, `name[occur][index1]` or `name[occur][index1][index2]`, where occur, index1 and index2 are all integers. The occur is the occurrence number, indicating the instantiation of the subcell in the current cell; and index1 and index2 are indices if a subcell has many copies organized as one or two dimensional array. Each input-pairs has the form "port-name, value", where "port-name" is an input-port name of the subcell, and "value" is the input value to that input port. Orders of these pairs are not important. Each output-pairs has the form "port-name, var", where "port-name" is an output-port name of the subcell, and "var" is a variable whose value should be set to that of the corresponding output-port after simulation of the subcell.

3.2.2 An Example

Here is an example of using C programming languages to describe a 16-bit parallel load register:

```
cell REG;
simulate C;

/* Main function for subcell REG: 16-bit Parallel Load Register.
 */
REG(phi1, phi2, S1, S2, Lin, Rin, A)
    in bit phi1, phi2;
    in bit S1, S2;
    in bit Lin, Rin;
    in bit A[16];
{
    out bit B[16], Lout, Rout;
    state R[18];         /* Variables of type "state" are static; they keep
                          * their values through across consecutive calls.
                          */
    int i;

    /* During phi1, things like loading and shifting will occur. */
    if (phi1 == 1)
    {
        if (S1 == 0 && S2 == 0)            /* hold */
            /* do nothing */;
        else if (S1 == 0 && S2 == 1)       /* shift left */
        {
            R[0] = Rin;
            for (i = 17; i; i--)
```

```
                R[i] = R[i-1];
        }
        else if (S1 == 1 && S2 == 0)        /* shift right */
        {
            R[17] = Lin;
            for (i = 0; i < 17; i++)
                R[i] = R[i+1];
        }
        else if (S1 == 1 && S2 == 1)        /* parallel load */
        {
            for (i = 0; i < 16; i++)
                R[i+1] = A[i];
        }

        /* Invalidate outputs. */
        for (i = 0; i < 16; i++)
            B[i] = X;
        Rout = X;
        Lout = X;

        return;
    }
    /* Validate the outputs. */
    for (i = 0; i < 16; i++)
        B[i] = R[i+1];
    Rout = R[0];
    Lout = R[17];
}
```

In the above program, the cell name is REG, and it has input ports phi1, phi2, S1, S2, Lin, Rin, and A[16]. Its output ports are B[16], Lout and Rout. The cell will be simulated at functional level, as indicated by the simulate clause.

3.2.3 The Preprocessor

For each subcell described at functional-level, our preprocessor will parse the C program, and produce another C program while making necessary changes, generate the .sim file from the extracted version of Magic layout by calling ext2sim, modify the .sim file by appending uses of subcells described at functional-level to it, and finally compile all the C programs produced, and link them with the simulator. If a module has more than one kind of description, e.g. both C program and Magic layout, the user has a choice of simulating

C program for speed or simulating Magic layout for accuracy. This is done with a change of a single word in the C program.

The C parser is written by using the UNIX parsing program generator YACC and the lexical analysis program generator LEX.

3.3 Interconnection Between Levels

The primary advantage of our multi-level simulator is that cells in different abstract levels can be simulated together. For instance, in our current implementation, a cell can contain any mixture of Magic layouts, schematic layouts, and C functions, as subcells. But how could a cell be included in another cell as a subcell?

This is straight forward if two cells are both Magic cells, or both are represented in schematic format. Both Magic and the schematic layout editor provide commands to include one cell into another cell as subcells. In fact, the schematic layout editor allows you to include cells in any representations as subcells.

If a Magic cell, say B, contains a C cell, say A, as its subcell, we can build a Magic cell for A containing nothing but input and output ports, and can use it as a complete Magic subcell in B. At simulation time, the corresponding C function for A will be called with current inputs, and its outputs will be produced by the C function, and the simulation of cell B continues.

A cell described in CHDL could include other cells as subcells, regardless of their representations, through map statement. But we encourage designers to use schematic layout for cells containing only subcells and their interconnections, since schematic layout is both easier to create and faster to be simulated. But if the cell has other logic described in CHDL besides subcells, the CHDL representation has to be adopted.

3.4 A Design Example

Msim is an interactive, event-driven, multi-level simulator for nMOS and CMOS transistor circuits. It accepts commands from the user, executing each command before reading the next.

The first step in using the simulator is to construct a circuit in either layout level or functional level, or both. The preprocessor must then be called to generate the `.sim` file, which is read to build the network in memory.

The next step usually involves setting input/output ports, defining clocks, and/or precharge some of the nodes. After input values have been established, their effect can be propagated through the network with commands described as follows. The `step` command allows input values to propagate through the network for n (default: 1) time units. The `cycle` command cycles n (default: 1) times through the clock, which is defined by the `clock` command. For each phase of the clock, the network is simulated until it stablizes. The `run` command have the same effect as the `step` command does, except that it runs until the network stablizes. The `cont` command continues the simulation for n (default: infinite) cycles. If n is not specified, the simulation will not stop until the network stablizes.

During the simulation, nodes could be traced by the command `trace`. Every time the state of a traced node is changed, the change will be reported to the user. States of nodes could also be displayed at any time with the command `print`.

Let's design a 4-bit binary counter from the beginning. Here are a few things that have to be done first:

- Choose a name for our counter, say *CTR*;
- Decide the functionality of the counter: increment every clock cycle if the control line

is high;

- Name the input/output ports, say *CLK* and *nCLK* for clocks, *T* for the control line, and *C*[4] for the four output bits;

Then we are ready to write done the description of our counter at functional level. Our first version may look like this:

```
cell CTR;       /* a binary counter */
bit T;
bit CLK, nCLK;
bit C[4];
simulate C;

/* ----- version 1 ----- */

CTR(T, CLK, nCLK)
    in bit T, CLK, nCLK;
{
    out bit C[4];
    state unsigned count = 0;
    int i;

    if ((T == 1) && (CLK == 1))
        count++;
    for (i = 0; i < 4; i++)
        C[i] = ((count & (01 << i)) ? 1 : 0);
}
```

Note that the above description contains no information whatsoever about how the counter is going to be implemented. All you need is a precise description that runs fast. Of course, if this was a larger circuit, you might very well want to simulate it at this time. In the next version, we begin considering how this counter will be constructed. How about four *T* flip-flops connected together serially. The following is the description of our *T* flip-flop at functional level.

```
cell TFLIP;       /* T type flip-flop */
bit T;
bit CLK, nCLK;
bit Q;
simulate C;
```

```
TFLIP(T, CLK, nCLK)
    in bit T, CLK, nCLK;
{
    out bit Q;
    state bit S = 0;

    if ((T == 1) && (CLK == 1))
        if (S == 0)
            S = 1;
        else if (S == 1)
            S = 0;
        else
            S = X;
    Q = S;
}
```

It is easy to see that the name of our *T* flip-flop is *TFLIP*, and it has a control line *T* and two clocks *CLK* and *nCLK* as inputs; and *Q* as output.

Having our *T* flip-flop as a subcell, the second version of out counter may contains more information on how the counter is constructed:

```
cell CTR;       /* a binary counter */
bit T;
bit CLK, nCLK;
bit C[4];
use TFLIP[4];
simulate C;

/* ----- version 2 ----- */

CTR(T, CLK, nCLK)
    in bit T, CLK, nCLK;
{
    out bit C[4];
    state unsigned count = 0;
    bit carry[3];

    carry[0] = (T & C[0]);
    carry[1] = (T & C[0] & C[1]);
    carry[2] = (T & C[0] & C[1] & C[2]);
    map TFLIP[0][0] (T, CLK, nCLK; C[0]);
    map TFLIP[0][1] (carry[0], CLK, nCLK; C[1]);
    map TFLIP[0][2] (carry[1], CLK, nCLK; C[2]);
    map TFLIP[0][3] (carry[2], CLK, nCLK; C[3]);
}
```

Again, if the circuit was larger, you should simulate it to find out any possible error in the specification or in the interconnection. You should simulate the whole counter, but you could also simulate the *T* flip-flop alone. After the verification, we proceed refining our *T* flip-flop, and produce the geometrical layout using Magic.

To verify the correctness of our layout, we can now simulate *CTR* at functional level while simulating the *T* flip-flop at switch-level. The only modification needed for this purpose is to changed the line `simulate C` to `simulate MAGIC` in the description file for *TFLIP* and use the second version of the *CTR*. Finally, the whole circuit of *CTR* is laid out using Magic, and the whole thing be simulated at switch-level. Its schematic representation is shown in Fig. 3.3.

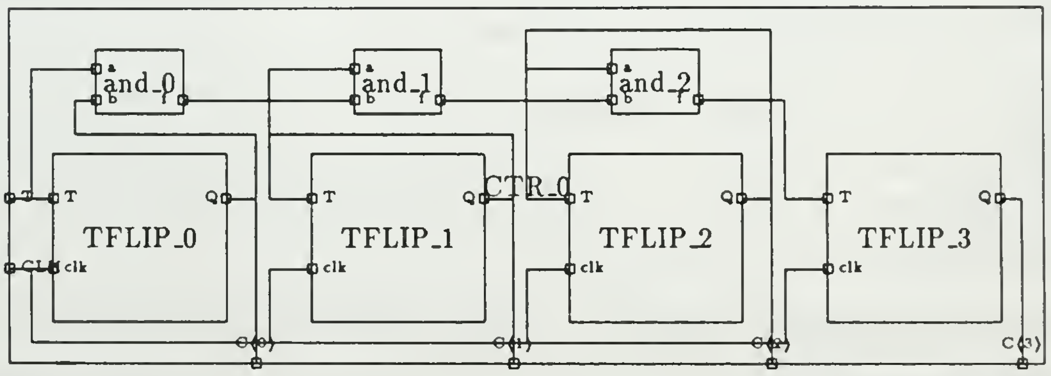


Figure 3.3: Schematic representation of a 4-bit binary counter

This is only a very small example, and sometimes you may think the descriptions at functional-level is unnecessary. It is not the case if you have a much larger circuit. In fact, you may need more than one version of functional-level descriptions, such as those for *CTR*. They are not redundant in the sense that the more abstract, non-constructive version of the description, usually produced at the beginning of the design, may still be useful at later stage. After verifying the correctness of the circuit, these non-constructive versions could be used for faster simulation.

Chapter 4

Automatic Placement

Automatic layout problem has been one of the most difficult as well as the most important problems encountered in the process of physical design of VLSI circuits. Many attentions have been attracted to this field in the last decade. It consists of two primary functions: mapping the circuit elements, which we will refer to as *basic modules*, onto locations on the layout surface, and interconnecting them according to a set of design rules [MeC 80]. The former is called *placement*, and the later is called *routing*. Although these two steps are highly related, we will concentrate on the placement problem only.

Good placement is a key step of automatic layout. It determines how much area the circuit will occupy, and the longest length as well as the total length of the interconnection wiring, which in turn limits the performance of the circuit. A poor placement could even make routing impossible. Although this suggests that the two steps should be done at the same time, and some researchers are actually doing that [Loo 79], the usually way is to consider routing as one of the key objectives while performing the automatic placement. The actually routing is done after the placement. One of the other important objective is to minimize the area taken by the modules.

In this part of the thesis, we will talk about automatic placement problems in custom design style, although some of the ideas could also be used in other design styles.

4.1 The Placement Problem

The placement problem consists of determining locations for basic modules in a circuit on the layout surface. Typically, a basic module represents a functional unit of the logic design such as an adder, a register, etc. , but in practice it could be anything. The input of the placement problem is a complete specification of a circuit's logical design and a number of requirements on the final layout. Here are some of the inputs:

- A set of basic modules;
- Shapes and aspect ratio allowances of each basic modules;
- List of nets that interconnect basic modules;
- Aspect ratio requirement for the final layout;

Net is another term for wire, which could be formally defined as a set of basic modules. A net $n = \{m_{i_1}, m_{i_2}, \ldots, m_{i_l}\}$ represents a wire that connects basic modules $m_{i_1}, m_{i_2}, \ldots, m_{i_l}$ all together. In this thesis, for simplicity, we only deal with nets that connect exactly two modules together. If the needs of nets connecting three or more modules arise, it could be replaced by a set of nets each connects only two modules. So the above net n could be replaced by a set of nets:

$$n_{jk} = \{m_{i_j}, m_{i_k}\}, \qquad 1 \leq j, k \leq l.$$

Another possible input is a list of *ports*, which forms the connection channel to the outside circuitry. Usually placement procedure is responsible for determining locations of these ports, but it is not discussed in this thesis. Instead, we assume that all ports are located at the "wiring center" (which will be defined later) of a module.

The goal of placement is to position basic modules on layout surface such that they do not overlap with each other, allow routing to be done without violating design rules, while optimizing any number of (usually conflicting) objectives imposed by the designer. Some common objectives include minimizing:

- layout area;
- wire length (longest or sum of);
- power dissipation;
- delay propagation;
- aspect ratio;
- a weighted sum of those above.

Unfortunately, placement problem with any one of the above as objective function results in an NP-hard problem [Don 80], [SaB 80]. Hence all attempts at solving the placement problem are heuristics.

4.2 Previous Works

Many heuristics have been used in solving placement problem. One of the most popular method is the *partitioning-based* placement [Lau 80], [She 89], [BHS 88]. The partitioning-based placement method essentially consists of a top-down mincut process, which divides both the layout surface and the set of modules into two halves. It then try to place one half of the modules on one half of the surface, and the other half of modules on the other half of the surface, recursively. The resulting floorplan is a slicing structure.

Another popular placement algorithm is *simulated annealing*. Starting from an initial (complete) placement, it trys to find a better one by making a slight change to the previous placement. By allowing transformation to a worse configuration occasionally, it avoids being trapped in local optima, hence capable of finding global optimal, at least in theory. Many automatic layout systems have adapted this method or its variations [MaG 88], [WoL 86], [Gro 87], [KlB 87], [GrS 84].

Force-directed placement follows a totally different way [For 87], [HWA 76]. It solves the placement problem by applying a physical model, where each pair of connected basic modules is considered to be joined by a spring whose stiffness is proportional to the number of connections between the modules. Then a simulation is done in which modules are allowed to move in response to the spring forces.

Noticing the amount of time consumed by previously used methods such as simulated annealing method, the *bottom-up iterative improvement* technique was introduced [MWL 87]. In this method, a layout is represented by a binary tree with leaves representing basic modules and internal notes specifying ways to combine two portions of a layout. The idea is to iteratively improve the current layout in a bottom-up fashion from the leaves to the root of the tree representing the layout. At each node, a set of heuristic rules are applied to find possible improvements. Our automatic placement system, which is going to be described in details in the next chapter, adapted this technique as its main algorithm.

Chapter 5

The Placement Generator

As part of our integrated system, a placement generator was developed. It is an interactive placement tool that could handle L-shape modules as well as rectangular shape modules. It has adopted similar method as used in [WoL 87] to deal with L-shape modules. The input of the placement generator consists of a complete logical specification of the circuit and some parameters that are part of objective functions to be optimized. Most of the parameters can be changed interactively. The algorithm used here is a mixture of bottom-up iterative improvement technique [MWL 87] and the simulated annealing technique. While bottom-up iterative improvement technique serves as the main part of the procedure, simulated annealing technique is only employeed to enable us to jump out of local optimals.

5.1 Layout Representations

In this section, we will describe how a layout is represented in the computer, what are the inputs to the placement generator, and how the quality of a layout is measured.

5.1.1 Circuit Specification

Circuit specification as part of the inputs of the placement generator consists of a set of n basic modules named $m_1, m_2, \ldots, m_n$. Each module is given an orientation which indicates the shape of the module. Rectangular shape modules have orientation of 0; L-shape modules have orientations from 1 to 4, as shown in Fig. 5.1.

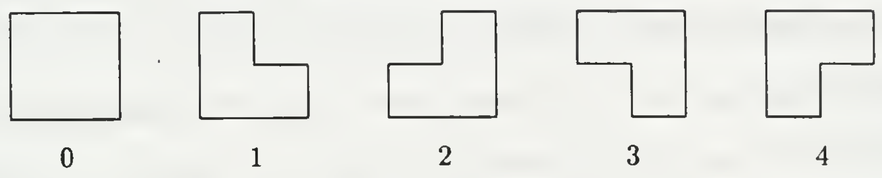


Figure 5.1: Orientations of modules

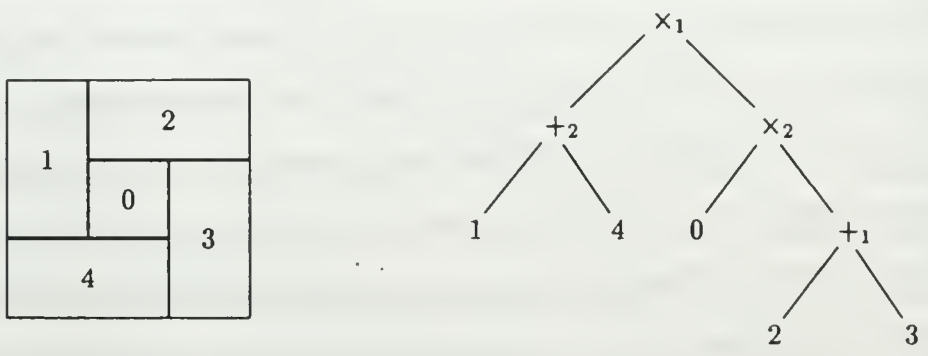
Dimensions are also provided for each module, which are fixed throughout the automatic placement process. We define the *wiring center* of a module to be the gravity center of its bounding box. It is easy to see that for a rectangular module, the wiring center so defined is just the center of its gravity, while for an L-shape module, the wiring center is the center of its bounding box. Wiring centers are used to compute approximations of wire lengths in the layout. The length of a wire from module m_i to module m_j is approximated to the Manhattan distance from wiring center of module m_i to that of module m_j in the placement generator. That is to say, all pins of a module are considered to be at its wiring center.

Modules are allowed to be rotated by multiple of 90°, flipped along X-axis or Y-axis, or any combination of them during the automatic placement.

Another part of the circuit specification is an $n \times n$ interconnection matrix $C = (c_{ij})_{n\times n}$, where c_{ij} are natural numbers. Here c_{ij} is the wiring density between module m_i and module m_j.

5.1.2 Placement Tree

We represent a layout by a binary tree called *placement tree.* Each leaf in a placement tree corresponds to a basic module. Each internal node represents a *super-module*, which consists of basic modules belonging to the subtree rooted at the internal node. Super-modules could also be of rectangular shape or one of the four L-shapes, just like basic modules. They could also be rotated by multiple of 90°, flipped along X-axis or Y-axis, or any combination of them during the automatic placement. The placement tree is just a way of specifying relative positions of modules. Fig. 5.2 shows a layout and its corresponding placement tree.



(a) Non-slicing structure (b) Placement tree

Figure 5.2: **A non-slicing** structure

The layout of a super-module is determined completely by the *operators* associated with each node in the subtree. **Two operators are associated with each internal** node, one binary and one unary. Only one operator, an unary operator, is associated with each leaf. There are four different *binary operators:* $\times_1$, $\times_2$, $+_1$, $+_2$; and there are five different *unary operators*: $\otimes_1$, $\otimes_2$, $\otimes_3$, $\oplus_x$, $\oplus_y$. Roughly speaking, $\times_1$ and $\times_2$ allow two modules to be positioned side by side, while $+_1$ and $+_2$ allow two modules to be positioned one on top of the other. $\otimes_1$, $\otimes_2$ and $\otimes_3$ rotate the module 90°, 180° and 270° respectively, while $\oplus_x$

and $\oplus_y$ flip the module along the X-axis and Y-axis respectively. See Appendix B for their detailed definitions.

Due to the ability of handling L-shape modules, certain layout which would be impossible if slicing structure is used, can be easily represented. Fig. 5.2 gives such an example.

5.1.3 Cost Function

Let d_{ij} be the Manhattan distance between the wiring centers of basic modules i and j, $1 \leq i, j \leq n$. The objective function optimized by the placement generator has the following format:

$$C = \alpha A + \beta W + \gamma R$$

where A is the area taken by the layout, W is the total wire length given by $\sum_{1 \leq i,j \leq n} c_{ij} d_{ij}$, and R is the aspect ratio of the layout. The three constants, α, β, and γ can be changed interactively during the automatic placement process. For instance, by choosing $\alpha = 1$, $\beta = 0$ and $\gamma = 0$, only the area taken by modules will be minimized.

There is a switch that let the designer to choose to optimize the longest wire length instead of the total wire length in the layout. So W in (5.1) could also represent the longest wire length in the layout if desired.

The cost of the layout may be changed every time we make an adjustment to the placement tree. This is because the placement tree determines locations of one module relative to that of the others. The time needed to compute total area A is constant, since at each node in the placement tree, we kept the dimensions of the module represented by the node. For the same reason, the aspect ratio R can also be computed in constant time. Time $O(n)$ is needed to compute total wire length W, though. This is because after each adjustment, we have to walk through the placement tree and recompute wiring centers of all the basic modules. Then the terms $c_{ij}d_{ij}$ are calculated again, where either one or both of the wiring centers of the basic modules m_i and m_j may have been changed.

5.2 Outline of the Algorithm

The algorithm starts by using cluster growth method to construct an initial placement tree. The objective of this initial placement is to construct a placement tree that combines the mostly connected modules first. In the next step it loops until some termination conditions are satisfied. During each iteration the algorithm does a post-order walk through the current placement tree, which we call *bottom-up refinement* procedure. At each node, all possible operators are tried and the one that results in the best *overall* layout is chosen for the node. At the end of each iteration, if a better layout is found, save it somewhere and display it on the screen; otherwise, the *simulated annealing* procedure is called, hoping to be able to jump out of the local optimal. At any time, the designer is free to interrupt the automatic placement process, and editing the current layout by hand though the graphics interface, and then resume the placement process.

In the following sections, we will discuss each of the procedures in details.

5.3 Initial Placement

Let's define the *connectivity* between two (possibly super) modules X and Y to be:

$$\sum_{i \in X, j \in Y} c_{ij} d_{ij}$$

where $i \in X$ means for all basic modules m_i that is part of the super-module X, and $j \in Y$ has similar meaning. The initial placement tree is constructed iteratively by combining smaller trees into bigger ones. Initially each basic module is in a separate tree consisting the module itself. Then the algorithm iterate until the forest becomes a single tree. During each iteration, two trees in the forest are found such that the connectivity between them is the maximum. These two trees are then replaced by a single tree which is obtained by combining the two. The binary operator at the root is chosen randomly, and no unary operator is given to the root at this time. Some kind of heuristics could be used here to choose good binary or unary operators for the root, but this is easily done though the first iteration of our bottom-up refinement procedure.

The time complexity of this step is $O(n^3)$, where n is the number of basic modules in the circuit.

5.4 Bottom-Up Refinement

In this phase of the algorithm, we walk through the placement tree in post order. For each node encountered, try all possible operators, binary (internal node only) and unary, and find the one that results in the best overall layout. Binary operator and unary operator can exist at the same time. So trying a new binary operator means trying it with the old unary operator. Similarly, trying a new unary operator means trying it with the old binary operator. Replace the old operator with the new one if they are different. After we have completed the whole walk, keep track of the best layout found so far; or if nothing has been improved, we call the simulated annealing procedure, which hopefully will make the next iteration a fruitful one by slightly "messing up" the current layout.

Although there can be at most one unary operator associated with each node in the placement tree, several unary operators can be applied to a node, having the effect of applying all of them, one after another. The way of doing it is to apply the first unary operator to the node, and changed the orientation and dimensions of the modules within the two children of the node, so that it looks just the same as if the fist operator has been applied. Eliminate the first operator associated with the node. Now we have space to apply the rest of the unary operators, in the same way. This makes application of any combination of unary operators possible.

During each iteration, we visit each node in placement tree exactly once. We then spend $O(n)$ time at each node, due to the computation of the cost function. So each iteration, i.e. each bottom-up refinement step, takes $O(n^2)$ time.

5.5 Simulated Annealing

Every time when the bottom-up procedure is completed, and nothing has been improved, this simulated annealing procedure will be called. The purpose of this procedure is to break out the local optimal so that the next time bottom-up procedure is called, hopefully some improvement will be achieved.

The simulated annealing procedure proceeds by first randomly choose a node, internal or leaf, from the current placement tree. It then randomly choose an operator for this node. Besides those binary and unary operators, two more operators may be applied to an internal nodes:

S_1: Swap the two children;

S_2: Swap one of its child with one of its grand-child;

After choosing the operator, replace the old operator with the new one, or perform the swap operation if it is S_1 or S_2. If this results in a better overall layout, make the change permanent; otherwise, still make the change permanent with probability:

$$e^{-\Delta C/T}$$

where ΔC is the difference in costs of the old and new layout, and T is the *temperature* which increases as more consecutive fruitless bottom-up iterations have been experienced.

This procedure will be repeated a number of times before we return to the bottom-up refinement procedure. The exact number of iterations is changing according to how well we have done in previous bottom-up refinement procedures, just like temperature does. But the range of this number is usually small, say from 20 to 30. If we consider this number to be a constant, then the time spend on simulated annealing procedure is $O(n)$, which is needed to compute the cost function.

5.6 Graphics Interface

A goal of the most automatic placement tools is to free designers from tedious details of physical VLSI circuit design. Due to the hardness of the placement problem, no matter how good a automatic placement tools is, there are some cases where the program just can't find a reasonably good layout within a reasonably short period of time. Keeping this in mind, we have included a graphics editing function in our system, which allows user to move modules *by hand* from one place to another, rotate them, or flip them. Users can stop the automatic placement process at any time, edit the current layout by hand, and then continue the automatic placement from the new configuration. This human intervention could sometimes save a lot of time.

5.7 An Example

We have run the automatic placement generator on the 16-bit CPU designed as an experiment under our CAD environment. Fig. 6.1 shows the schematic layout produced by hand. And Fig. 5.3 shows the result generated by the automatic placement generator, without any human intervention. Note that all ports are considered by our placement tool to by located at the center of each module. Also, no routing space is left around modules; the modules should contain routing spaces themselves.

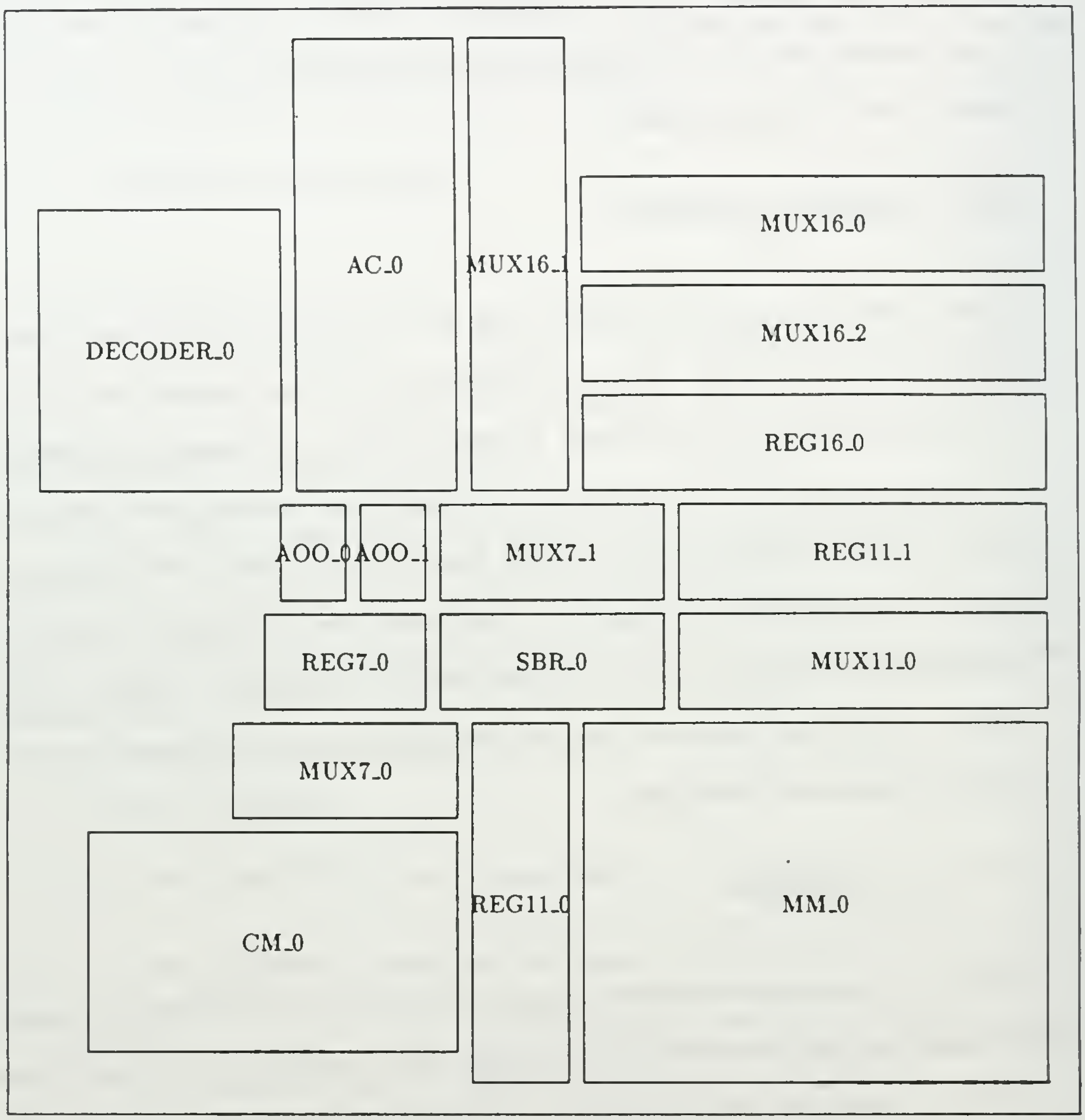


Figure 5.3: Placement of a CPU generated by the placement generator

Chapter 6

The Schematic Layout Representation

Currently, there are two levels at which a circuit could be specified in order for it to be simulated under our environment, the CHDL, or the Magic layout format. As another level between these two, we have developed the schematic layout representation. While CHDL is used primarily for behavioral descriptions of circuit, the schematic layout representation is intented for structural descriptions. From designer's point of view, schematic layouts are much easier to create than Magic layout; it also makes the creation of HDL descriptions much easier. Actually, the creation of a schematic layout of what a designer wants to design is a good starting point. It provides the designer with a general, clear picture of the whole circuit in a very short period of time.

One of the biggest problems with hardware description languages is the specification of interconnections between modules. If a circuit contains lots of modules, each with a number of ports, it is very difficult to specify which port are connected to which. Conventional methods, such as representing a net as a list of port names, suffer the problems like hard to create, hard to read, and fragile to modifications. With the help of a graphics editor, interconnections between modules could be specified and modified by clicking of buttons. The graphics picture of nets as line segments are also much easier to read and less erroneous.

6.1 Circuit Representation

At schematic level, a module in a circuit is represented as a rectangle on the screen. Its size and shape could be considered to be estimations of the size and shape of the final Magic layout. It could be meaningless too, depending whether or not the designer chooses to use this information later. Input and output ports of a module are represented as a small square on the boundary of the module. Again, its position could be an estimation of its final location in the module's final Magic layout, or it could be meaningless. Nets, defined as a list of ports, are represented by a list of line segments connecting those squares representing the ports. Ascii names could be assigned to both modules and ports, but not nets. Fig. 6.1 gives an example of a 16-bit CPU designed in schematic layout format.

Modules are organized in a hierarchical way – modules could contain submodules. Basic modules – modules that do not contain any submodules – usually contains a piece of C code describing its logical behavior. Only one copy of a module will be kept, even if it is used, as submodules, more than once in the hierarchy.

Modules described in schematic layout are considered to be at lower level than those described in C, since they contain more information than C modules do, such as geometrical dimensions and shapes. Meamwhile, they are considered to be at higher level than those described in Magic layout, since they obviously contains less information than Magic modules do. For instance, they do not have any transistors, like Magic modules do.

6.2 The Schematic Layout Editor

A menu-driven tool, called *schematic layout editor*, is provided to allow users to create and manipulate modules, ports, and nets in almost anyway they wants. Using the editor, modules could be created, moved, resized, rotated, or flipped in very much the same way

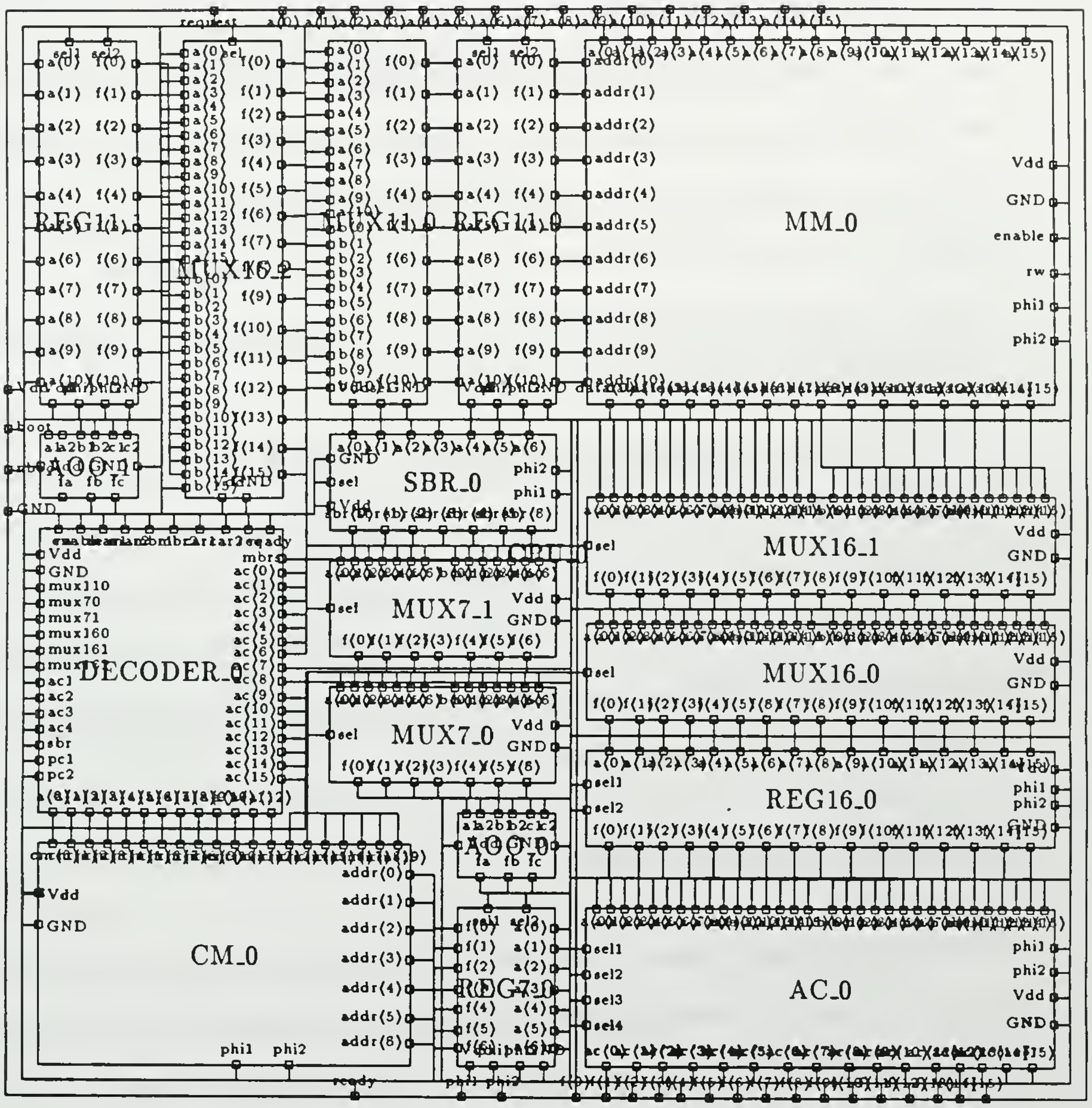


Figure 6.1: Schematic Layout of a 16-bit CPU

as in most graphics editors. Ports could also be created or moved. Nets are created by clicking buttons at those ports that should be connected. A different button signals end of the list, and an automatic router will be called to draw line segments connecting the ports.

As mentioned before, the schematic layout of a circuit is organized in a hierarchical way, with modules containing submodules. Designers could walk up and down through the hierarchy tree, and modify the layout. Since only one copy is kept for each module, a modification will affect all occurrences of the module, should it be used many times in the hierarchy. At leaves of the hierarchy tree, all modules are basic modules. A text editor, such as "vi", could be invoked directly from the schematic layout editor to compose a piece of CHDL program for the module.

The information contained in a schematic layout could be extracted, and then simulated, together with behavioral specifications of basic modules, using *Msim*. This not only eliminates the needs to manually specify interconnections between submodules using the "map" statements, but also more efficient in terms of simulation speed. The placement generator previously described could also be invoked to optimize the schematic layout in terms of area estimation or wire length, or both.

6.3 Routing Algorithm

As part of the schematic layout editor, we have implemented an automatic router. It allows user to draw nets by pointing out list of ports only, and the router will connect the ports for the user. Since the main purpose is to allow designers to specify interconnections in an efficient manner, the automatic router does only global routing. Wires are running through the middle of routing channels, and no channel routing is done. To have a clear picture of which ports are connected to which, nets can be highlighted, one at a time. The method used by the automatic router is described in the following algorithm.

Algorithm 6.1 *Global Routing Algorithm.*

INPUT: *A list of rectangles as modules; and a list of ports, as nets, on the boundaries of rectangles.*

OUTPUT: *A list of line segments connecting the ports in the given list.*

METHOD:

1. *Divide routing space into "channels" by extending boundaries of modules. Each channel is of rectanglar shape.*
2. *Combine small channels into large ones as much as possible, as long as they are of rectanglar shape.*
3. *Create an undirected graph G with vertices representing channels, and with edges representing adjacency of channels. Two vertices will be connected iff the two channels represented by the two vertices are adjacent in the layout.*
4. *Do a breadth-first search on G, and find the minimum spanning tree containing channels in which one or more ports on the given list is located.*
5. *Connect ports on the list by going through those channels on the minimum spanning tree. Wires go from one channel to another through the middle point of their common boarder.*

□

Chapter 7

A Design Example

For the purpose of demonstration, we have designed a microprocessor under our CAD environment. The configuration and microinstruction format are from [Man 82]. It is a 16-bit microprocessor controlled by a microprogram stored in a control memory. For simplicity, there will be no operating system. As soon as the system is booted, it starts loading a batch job from a card reader, and then performs the job. There is be a line printer available for user jobs to print out their results. The card reader is also available to user jobs.

Its design started with a schematic layout, created using our schematic layout editor. The top level contains three modules, CARDREADER, PRINTER, and CPU. At the second level, CPU is decomposed into seventeen submodules, all of them are currently basic modules. Fig. 7.1 shows the final layout produced by the editor.

Functional behaviors of basic modules were then specified using the CHDL. No behavioral specification is written for CPU, since we have already had a very clear decomposition of it. As an example, we have given below a complete listing of the behavioral specification for the accumulator AC.

```
cell AC;
out bit ac[16];
in bit b[16];
in bit a[16];
in bit sel1;
```

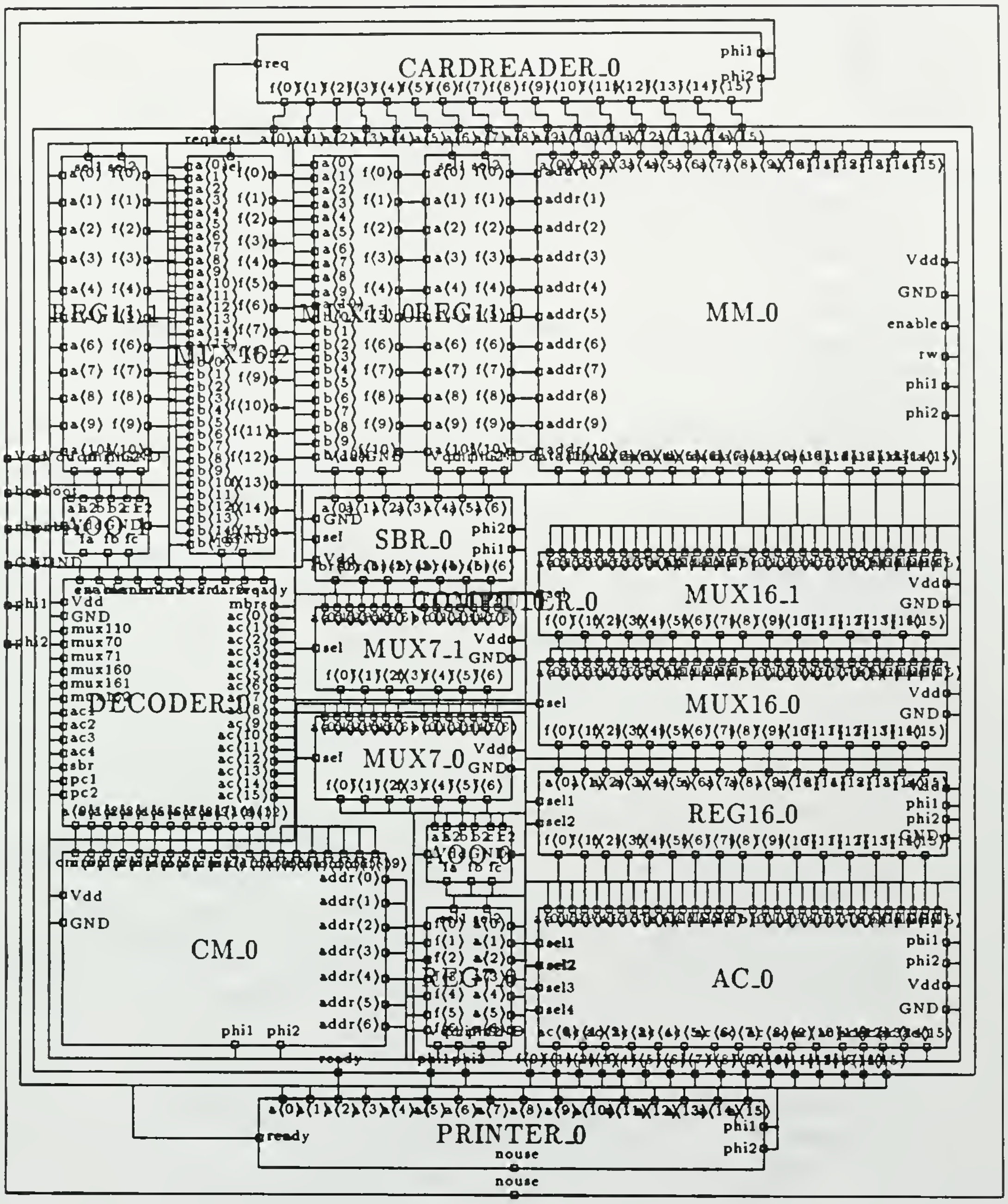


Figure 7.1: Schematic Layout of a 16-bit Microprocessor

```
in bit sel4;
in bit sel3;
in bit sel2;
in bit phi1;
in bit phi2;
in bit Vdd;
in bit GND;
simulate C;

AC(b, a, sel1, sel4, sel3, sel2, phi1, phi2, Vdd, GND)
    in bit b[16];
    in bit a[16];
    in bit sel1;
    in bit sel4;
    in bit sel3;
    in bit sel2;
    in bit phi1;
    in bit phi2;
    in bit Vdd;
    in bit GND;
{
    bit msim_tmp;
    state out bit ac[16];
    state bit f[16];
    int aa, bb, cc;
    unsigned int t1, t2, byteToInt();
    unsigned int sel;
    int i;

    if (phi1 == 1)
    {
        t1 = byteToInt(a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7]);
        t2 = byteToInt(a[8], a[9], a[10], a[11], a[12], a[13], a[14], a[15]);
        aa = (t1 << 8) | t2;
        t1 = byteToInt(b[0], b[1], b[2], b[3}, b[4], b[5], b[6], b[7]);
        t2 = byteToInt(b[8], b[9], b[10], b[11], b[12], b[13], b[14], b[15]);
        bb = (t1 << 8) | t2;
        sel = byteToInt(0, 0, 0, 0, sel1, sel2, sel3, sel4);

        switch (sel)
        {
            /* Arithmetic Micro-Operations */
            case 0:     /* ac <- a + b (0000) */
                cc = aa + bb;
                break;

            case 1:     /* ac <- a - b (0001) */
                cc = aa - bb;
                break;

            case 2:     /* ac <- a + 1 (0010) */
                cc = ++aa;
        break;

    case 3:      /* ac <- a - 1 (0011) */
        cc = --aa;
        break;

    case 4:      /* ac <- complement of a (0100) */
        cc = ~aa;
        break;

    case 5:      /* ac <- complement of a + 1 (0101) */
        cc = ~aa + 1;
        break;

    case 6:      /* ac <- a + complement of b (0110) */
        cc = aa + ~bb;
        break;

    case 7:      /* ac <- a + complement of b + 1 (0111) */
        cc = aa + ~bb + 1;
        break;

    /* Logic Micro-Operations */
    case 8:      /* ac <- 0 (1000) */
        cc = 0;
        break;

    case 9:      /* ac <- a XOR b (1001) */
        cc = aa ^ bb;
        break;

    case 10:     /* ac <- a AND b (1010) */
        cc = aa & bb;
        break;

    case 11:     /* ac <- a OR b (1011) */
        cc = aa | bb;
        break;

    case 12:     /* ac <- b (1100) */
        cc = bb;
        break;

    case 13:     /* NOP (1101) */
        cc = byteToInt(f[0], f[1], f[2], f[3], f[4], f[5], f[6], f[7]);
        cc = (cc << 8) |
            byteToInt(f[8], f[9], f[10], f[11], f[12], f[13],
                      f[14], f[15]);
        break;

    case 14:     /* ac <- shift left of a (1110) */
        cc = aa << 1;
```

```
                break;

            case 15:    /* ac <- shift right of a (1111) */
                cc = aa >> 1;
                break;

            default:
                break;
        } /* switch */

        for (i = 0; i < 16; i++)
            f[15-i] = (cc >> i) & 01;
    }
    else if (phi2 == 1)
    {
        for (i = 0; i < 16; i++)
            ac[i] = f[i];
    }
}
```

The schematic layout and behavioral specifications were then simulated using the multi-level simulator. After the successful simulation, we started refining modules by creating geometry layout for them, using the Magic layout editor. The whole system was simulated after every refinement to ensure its correctness. Also, the automatic placement tool described in Chapter 5 was used to optimize the layout, and the result is shown in Fig. 5.3.

7.1 Microprocessor Configuration

As shown in Fig. 7.2, the system consists of the following key components:

- Accumulator Register AC — a 16-bit register capable of performing 16 operations on its two 16 bit input. Its result is also a 16-bit number.
- Memory Module MM — a 2048 × 16-bit RAM with an address bus of 11 bits and a data bus of 16 bits.
- Memory Address Register MAR — a 11-bit register providing addresses for MM.

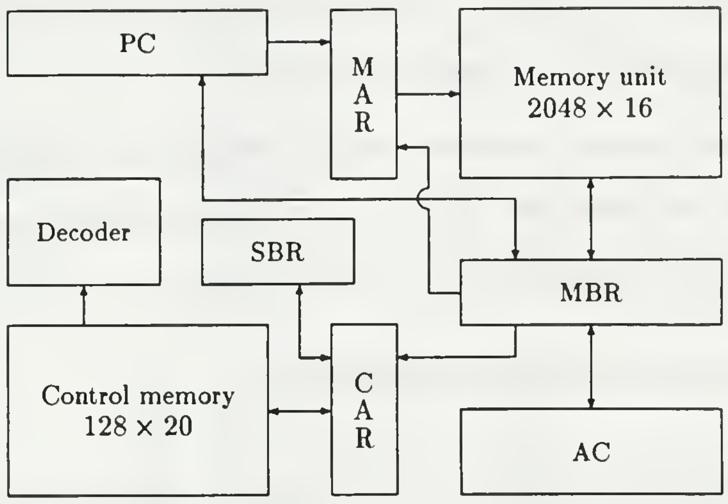


Figure 7.2: Block diagram of the microprocessor

- Memory Buffer Register MBR — a 16-bit register providing and/or receiving data for/from the MM.
- Program Counter PC — a 11-bit register which serves as the program counter.
- Control Memory CM — a 128 × 20-bit ROM in which stores the microinstructions controlling the microprocessor. As MM, its has its own address register register.
- Control Memory Address Register CAR — a 7-bit register providing addresses for CM.
- Subroutine return register SBR — a 7-bit register which stores the return address when a subroutine is called in the microprogram.
- Decoder — a module that interprets the control bits of the encoded microinstructions, and controls the behaviors of other modules.
- Card Reader CARDREADER — which simulates a card reader.
- Printer PRINTER — which simulates a line printer.

One thing that is not shown in the diagram of Fig. 7.2 is the control signals going from Decoder to all the other components of the system.

The system employs a two phase clocking. During clock phase *phi1*, all the inputs are valid; during clock phase *phi2*, outputs of all modules are changed, and no input should be considered valid. All registers are master-slave registers.

7.2 Microinstruction Format

Each microinstruction residing in the control memory consists of 20 bits divided into four functional fields, as shown in Fig. 7.3. The numbers on top of each field are the number of bits contained in that field.

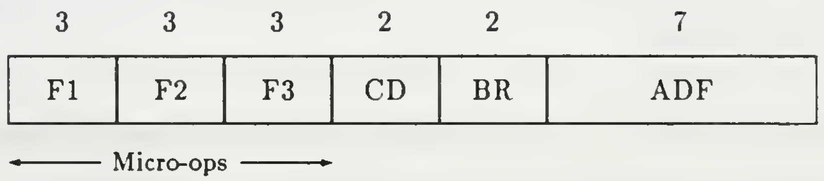


Figure 7.3: Microinstruction format (20 bits)

The *micro-ops* field specifies micro-operations for the microprocessor. The *CD* field selects status bit conditions, the *BR* field specifies the type of branch, and the *ADF* field contains an address.

The micro-ops field is subdivided into three subfields F1, F2, and F3, of three bits each. The three bits in each field are encoded to specify seven distinct micro-operations as listed in Table 7.1. This means a maximum of three micro-operations could be performed during each clock cycle.

The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table 7.2. The BR (branch) field consists of two bits, as shown

F1	Micro-operation	Symbol
000	None	NOP
001	$AC \leftarrow AC + MBR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow MBR$	BRTAC
101	$MAR \leftarrow MBR(AD)$	BRTAR
110	$MAR \leftarrow PC$	PCTAR
111	$M \leftarrow MBR$	WRITE

F2	Micro-operation	Symbol
000	None	NOP
001	$AC \leftarrow AC - MBR$	SUB
010	$AC \leftarrow AC \vee MBR$	OR
011	$AC \leftarrow AC \wedge MBR$	AND
100	$MBR \leftarrow M$	READ
101	$MBR \leftarrow AC$	ACTBR
110	$MBR \leftarrow MBR + 1$	INCBR
111	$MBR(AD) \leftarrow PC$	PCTBR

F3	Micro-operation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus MBR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$shlAC$	SHL
100	$shrAC$	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow MBR(AD)$	BRTPC
111	Reserved	—

Table 7.1: Micro-operation fields bit assignment

in Table 7.3. It is used, in conjunction with the address field ADF, to choose the address of the next microinstruction.

CD	Condition	Symbol	Comments
00	1	U	Unconditional (always = 1)
01	$MBR(I)$	I	Indirect address bit
10	$AC(S)$	S	Sign bit of AC
11	$AC = 0$	Z	Zero value in AC

Table 7.2: CD (condition) field bit assignment

7.3 Assembly Language

Our machine language consists of 16 bits per instruction. The assembly instruction format is show in Fig. 7.4. The numbers on top of each field are the number of bits contained in that field.

BR	Symbol	Function
00	JMP	$CAR \leftarrow ADF$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow ADF, SBR \leftarrow CAR + 1$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2-5) \leftarrow MBR(OP), CAR(1) \leftarrow 0$,
		$CAR(6-7) \leftarrow 1$ (Micro-operation mapping)

Table 7.3: BR (branch) field bit assignment

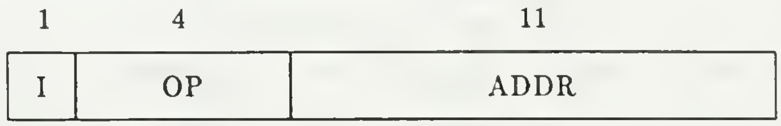


Figure 7.4: Assembly instruction format (16 bits)

Since we have a 2048×16 bits of memory, we need 11 bits for memory reference in our instructions. The last 11 bits are used for this purpose. Putting aside one bit (the first bit) as indirect addressing bit, we have four bits left for operation specifications. This indicates the maximum of 16 assembly instructions. Since some of the instructions do not need the indirect bit (they do not make memory references), we have assigned new meanings to such instructions with indirect bit on. This allows us to have more than 16 assembly instructions. Table 7.4 shows all the instructions, their binary codes, and a briefly description for each of them. In the table, m represents the address field of the instruction. If the I field of an instruction in the table is 0 (or 1), it means this field must be 0 (or 1) for this instruction; if the field is blank, either 1 or 0 could be put there, and it is significant.

In the following section, we will have the microprogramming implementation of these instructions. As an example, here is the assembly-language program that loads user jobs from card reader every time the system is booted. It is stored from location 0 through 15 of the memory module. The ORG (origin) instruction in the list is a pseudo-instruction

Symbol	I	OP	Description
ADD		0000	Add M to AC
SUB		0001	Subtract M from AC
AND		0010	And (bitwise) M to AC
OR		0011	Or (bitwise) M to AC
XOR		0100	Exclusive-Or M to AC
LOAD		0101	Load M into AC
STORE		0110	Store AC into M
CIR	0	0111	Circulate right AC, fill leftmost with 0
RET	1	0111	Branch *indirectly* to m
CIL	0	1000	Circulate left AC, fill rightmost with 0
INC	0	1001	Increment AC
HALT	1	1001	Halt computer
JZA		1010	Jump to m if AC is Zero
JNA		1011	Jump to m if most significant bit of AC $= 1$
JMP		1100	Branch unconditionally
CALL	0	1101	Save return address in m and jump to $m+1$
IN	0	1110	Read from card-reader into AC
CLA	1	1110	Clear AC
CMA	0	1111	Complement AC
OUT	1	1111	Write AC to line printer

Table 7.4: Assembly Instructions

that indicates the memory location in which the following instruction or operand will be put. We use this same pseudo-instruction in our microprograms as well. Also note that everything after and includes two consecutive backslashes are comments.

```
        ORG 0
LOOP:   IN                          // read from card reader
        STORE   I INDRCT            // store
        INC                         // increment the word just read
        JZA       USER              // if all 1's was read in, done
        LOAD      INDRCT            // update the indirct address
        INC
        STORE     INDRCT
        JMP       LOOP              // go to read next instruction

ORG 15
INDRCT: 0000 0000 0001 0000         // starting address where user job is loaded

        ORG 16
USER:                               // user job starts here
```

7.4 Contents of Control Memory

When the microprocessor is booted, both the program counter PC and the control memory address register CAR will be set to zero. Then the next thing the microprocessor is going to do will be the execution of the microinstruction residing at location 0 of the control memory. So we have arranged the control memory such that the microinstructions performing the fetch cycle is located at address 0. Them are followed by a list of microinstructions groups, four in each group. Usually, each such group implements one assembly instruction. Some groups implement two assembly instructions, and they are distincted by the different values in the indirect field. The following is the contents of our control memory, which implements our assembly instructions.

```
        ORG 0
FETCH:  PCTAR            U JMP  NEXT
        READ, INCPC      U JMP  NEXT
        BRTAR            U MAP

        ORG 3
ADD:    NOP              I CALL INDRCT
        READ             U JMP  NEXT
        ADD              U JMP  FETCH

        ORG 7
SUB:    NOP              I CALL INDRCT
        READ             U JMP  NEXT
        SUB              U JMP  FETCH

        ORG 11
AND:    NOP              I CALL INDRCT
        READ             U JMP  NEXT
        AND              U JMP  FETCH

        ORG 15
OR:     NOP              I CALL INDRCT
        READ             U JMP  NEXT
        OR               U JMP  FETCH

        ORG 19
XOR:    NOP              I CALL INDRCT
        READ             U JMP  NEXT
        XOR              U JMP  FETCH
        ORG 23
LOAD:   NOP             I CALL INDRCT
        READ            U JMP  NEXT
        BRTAC           U JMP  FETCH

        ORG 27
STORE:  NOP             I CALL INDRCT
        ACTBR           U JMP  NEXT
        WRITE           U JMP  FETCH

        ORG 31
CIR:    NOP             I JMP  NEXT+1
        SHR             U JMP  FETCH
RET:    NOP             U CALL INDRCT
        BRTPC           U JMP  FETCH

        ORG 35
CIL:    SHL             U JMP  FETCH

        ORG 39
INC:    NOP             I JMP  NEXT+1
        INCAC           U JMP  FETCH
HALT:   HALT

        ORG 43
JZA:    NOP             Z JMP  NEXT+1
        NOP             U JMP  FETCH
        NOP             I CALL INDRCT
        BRTPC           U JMP  FETCH

        ORG 47
JNA:    NOP             S JMP  NEXT+1
        NOP             U JMP  FETCH
        NOP             I CALL INDRCT
        BRTPC           U JMP  FETCH

        ORG 51
JMP:    NOP             I CALL INDRCT
        BRTPC           U JMP  FETCH

        ORG 55
CALL:   PCTBR, BRTAC    U JMP  NEXT
        WRITE, ACTBR    U JMP  NEXT
        INCBR           U JMP  NEXT
        BRTPC           U JMP  FETCH

        ORG 59
CLA:    CLRAC           I JMP  FETCH
IN:     INPUT1          U JMP  NEXT
        INPUT2          U JMP  NEXT
        BRTAC           U JMP  FETCH
        ORG 63
CMA:    NOP             I JMP  NEXT+1
        COM             U JMP  FETCH
OUT:    OUTPUT          U JMP  FETCH

        ORG 123
INDRCT: READ            U JMP  NEXT
        BRTAR           U RET
```

Note that the space between 66 and 122 are empty. In fact, by careful arrangement, we can pack these microinstructions into a 64 × 20-bit ROM.

7.5 Simulation Results

As we have mentioned previously, our microprocessor will start loading a program from the CARDREADER as soon as it is booted. Upon finishing the loading, it jumps to the start of the program just loaded, and begins the execution. We have tested several programs. The program listed below does multiplication of two unsigned 8-bit binary numbers. The result will be a 16-bit binary number. The subroutine ADD could be expanded into the main program, since it is called from only one place. We choose to implement it as a subroutine to demonstrate the way subroutine calls works. The subroutine call is implemented by storing return address at the first location of the subroutine, and then jump to the second location of the subroutine. The return instruction is implemented by jumping *indirectly* to the beginning of the subroutine, where the return address is stored by the caller.

```
        ORG 16
LOOP:   LOAD    N1
        AND     MASK
        JZA     L
        CALL    ADD
L:      LOAD    N1
        CIR
        STORE   N1
        LOAD    N2
        CIL
        STORE   N2
        LOAD    CTR
        INC
        STORE   CTR
        SUB     N8
        JZA     DONE
        JMP     LOOP
DONE:   LOAD    P
        OUT
        HALT

ADD:    0       // return address
        LOAD    P
        ADD     N2
        STORE   P
        RET

        // data
MASK:   0000 0000 0000 0001       // mask to get last bit
N1:     0000 0000 1000 1010       // multiplicand
N2:     0000 0000 0110 1010       // multiplier
P:      0000 0000 0000 0000       // product
N8:     0000 0000 0000 1000       // number 8
CTR:    0000 0000 0000 0000       // counter to remember #bits done
END:    1111 1111 1111 1111       // program end marker
```

To demonstrate the performance of our system, we have simulated the microprocessor using several test programs, including the above multiplication program. The test was conducted on a Sun workstation. Different combinations of description levels for modules and submodules were tried out, and their results are shown Table 7.5.

test program	*case1*	*case2*	*case3*
multiplication	181.3	249.3	890.8
division	203.1	296.7	931.0
fibonacci	219.7	301.5	971.2

Table 7.5: Performance comparasion

In Table 7.5, the three cases means:

case1: All modules are simulated at functional-level;

case2: Only REG7 is simulated at switch-level.

case3: The following modules are simulated at switch-level: AOO, REG7, REG11, REG16, MUX7, MUX11, MUX16. The rest are simulated at functional-level;

The microprocessor starts by running a loader which loads in a test program. It then transfers the control to the test program just loaded in. One thing that is clearly shown in Table 7.5 is that functional-level simulation is much faster than switch-level simulation. It tells us that we should take this advantage by simulating at switch-level only those modules that are newly designed. This is, as soon as we have fully tested a module, we should change to simulate it at functional-level while testing other modules.

Chapter 8

Conclusions And Future Works

8.1 Conclusions

We have designed and implemented an integrated VLSI CAD environment that supports almost all phases of the VLSI circuit design cycle, from high-level circuit description down to mask generation. Tools that have been integrated under the environment include a multi-level simulator, an automatic placement tool, a schematic layout editor, and the geometrical layout editor Magic developed at UC Berkeley.

Experience shows that the design productivity can be greatly increased under the environment. As an experiment, I implemented a basic 16-bit microprocessor using the system. It took me less than a week to produce the working version of the schematic layout and the functional descriptions of the basic modules. This would be impossible if we had to layout everything in Magic. Also, it would be much more difficult if we had to specify the 267 nets via "map" statements.

8.2 Future Works

Our system has demonstrated many of its advantages. But it is far from complete. Many nice features and tools are still waiting to be added to the system. The following is a partial list.

8.2.1 Extensions

The schematic layout editor has made the specification of interconnections between modules very easy. Nets are created by clicking buttons at those ports that should be connected. There are times when we want to connect array of ports, say `a[32]`, with another array of ports, say `f[32]`, in such a regular way that `a[i]` will be connected with `f[i]` where i goes from 0 to 31. In this case, it would be much easier if we could issue a command: `connect a[32] f[32]`, and the editor will automatically connect `a[0]` with `f[0]`, `a[1]` with `f[1]`, ..., and `a[31]` with `f[31]`.

Another useful command for the schematic layout editor would be to expand a subcell into an array (one dimensional or two dimensional) of subcells. And the `connect` command mentioned above would have another dimension of power.

Current implementation of the schematic layout editor includes a built-in automatic router which does global routing. Designers specify nets by clicking mouse buttons at those ports being connected, and the automatic router will do the routing and draw the virtual wires. Although the main purpose of the router is to allow designers to specify interconnections easily, it could also serve as a guideline for physical routings at later stage of the design. Toward this end, the automatic router should consider information such as the density of routing channels. It should also provide graphics interface allowing designers to edit certain critical paths manually.

By collecting some extra information while simulating circuit, the multi-level simulator could provide facilities to conduct architectural-level simulations. For example, the simulator could keep track of the number of times a module was activated, which could then be used by designers to predict performance of the system, or to determine architectural parameters of the system. Furthermore, the simulator could keep track of the number of times a node was changed, which could be used by designers to do performance analysis at circuit-level.

In our current implementation, all nodes at functional level are considered input nodes. Signals coming in through input ports will have the strength ω. Also, signals going out through output ports will have the strength ω. While this assumption simplifies the CHDL language, it also loses accuracy at some degree. Once we have refined the cell into Magic layout, say, the assumptions may become incorrect. Some errors that would not otherwise exist may occur. To deal with this problem, we can easily introduce notions to specify signal strengths at functional level, and let our simulator Msim take them into account.

The switch level simulation could be speed up if nodes are stored in memory by static connected components of the ternary switch graph, hence minimizing page fault.

It would be more accurate if the automatic placement tool could consider ports on boundaries of modules, instead of at centers of modules. Automatic routing tools could be added to perform global routing and channel routing. Also, the automatic placement tool should be able to operate directly on the schematic layout.

8.2.2 Debugging Toolkit

For the purpose of debugging, the multi-level simulator should be able to report the causes of changes of a net when asked. Even better, imagine all reports from the simulator were graphical, such as on the schematic layout, or on the Magic layout. The problem would be pinned out in seconds.

Designers should be able to place break points anywhere in the circuit, at any level. Designers should also be able to construct a boolean expression in a general form, for a break point, and ask the simulator to stop the simulation conditionally. For example, the simulator could be asked to stop if a specific node becomes 1; or if one of the inputs of a specific module is changed.

8.2.3 New Tools

New hardware description languages at functional level, such as VHDL, could be added to the system. Other circuit representations at intermediate levels, such as gate level, would certainly make refinement conversions smoother.

The multi-level simulator operates only in unit-delay mode. Timing information is completely ignored. Although we have tools such as Crystal for timing analysis of Magic layout, it would be hard to do any such analysis on circuit with components described at higher levels. For this purpose, we need a mechanism in our HDL to specify timing information for those modules whose Magic layout are not available yet. It would be better if our multi-level simulator could also use timing information to operate in linear delay mode.

The technique of compiling circuit layout into low level machine code in order for them to be simulated, as did in [BBB 87] and [WHP 87] may be explored for the switch-level simulation of Msim. For each module, several relocatable object codes are produced, one from circuit layout and one from functional specification, say. They are then selectively linked together with other modules, depending on which level we want each module to be simulated at.

The power of our system would be greatly enhanced if it could provide interface with other existing systems. For instance, circuit representations at functional-level and schematic-level may be transformed into other formats so that a silicon compiler such as Flamel [Tri 85] could be used to produce VLSI chips directly. Another possibility would be to transform current circuit representations into that acceptable by other simulation engines such as the Yorktown Simulation Engine [Pfi 82], [Den 82], [KrP 82].

Appendix A

Summary of Msim Commands

Each Msim command has the following simple syntax:

$$\text{cmd } arg_1 \; arg_2 \; \ldots \; arg_n \; \langle \textit{Return} \rangle$$

where "cmd" specifies the operation to be performed and the arg_i are arguments needed for that operation. The arguments are separated by spaces (or tabs) and the command is terminated by a ⟨*Return*⟩. The command name can be abbreviated, just as long as you type enough characters to distinguish it from all other commands.

Many of the Msim commands take node names as parameters. The nome of a node is the label attached to it in the Magic layout, or in the schematic layout, prefixed by the complete path name consisting of the name of parent cell, grandparent cell, and so on, divided by backslashes. If two nodes with different labels are connected together, either name can be used to reference the connected node.

The following is the list of commands with their syntax and semantics.

# *comment ...*

Lines beginning with # are treated as comments and are ignored. It is useful for comments or temporarily disabling certain commands in a command file.

?

Print a command summary. Same as the **help** command.

alias *node* $name_1$ $name_2$ ...

Define $name_1$, $name_2$ ..., to be the nicknames of the node *node*, so that they can be used to reference the same node. This is especially useful for nodes with a very long name, such as nodes of a subcell deep down the hierarchy tree of the circuit.

clock *node* phi_1 phi_2 ...

Define *node* to be one of the clocks in the circuit, with phi_i as its value during the i^{th} phase.

cont [#*steps*]

Continue the simulation from wherever it has been left for #*steps* steps if provided. Otherwise continue until stabilize.

cycle [#*cycles*]

Simulate the circuit for #*cycles* cycles.

exit

Exit Msim and return to system. Same as the **quit** command.

help

Print a command summary. Same as the ? command.

high $node_1$ $node_2$ ...

Set a list of nodes, $node_1$ $node_2$ ..., to logic value 1, with strength ω.

load $file_1$ $file_2$ ...

Load circuit descriptions from files $file_1$ $file_2$ ....

low $node_1$ $node_2$ ...

Set a list of nodes, $node_1$ $node_2$ ..., to logic value 0, with strength ω.

merge *$node_1$ $node_2$* ...

Electronically connect the nodes *$node_1$ $node_2$* ....

mode [*race* | *norace*]

Set the simulation mode to either detect-race-condition or to do-not-detect-race-condition. Without argument, the mode will be flipped.

phigh *$node_1$ $node_2$* ...

Set a list of nodes, *$node_1$ $node_2$* ..., to logic value 1, with strength κ_2, which is the strongest among storage nodes.

plow *$node_1$ $node_2$* ...

Set a list of nodes, *$node_1$ $node_2$* ..., to logic value 0, with strength κ_2, which is the strongest among storage nodes.

print [*?* | *keyword* | *$node_1$ $node_2$* ...]

If a list of node names, *$node_1$ $node_2$* ..., is given as arguments, information about these nodes, such as their logic values, will be printed out. If *?* is given as argument, the list of valid keywords with a brief description of their meanings will be printed out, which has the following meanings.

all Print out information of *all* nodes.

cells Print out information of all *cells*.

clocks Print out information of all defined *clocks*.

events Print out contents of the *event* queue.

inputs Print out current list of *input* nodes.

traced Print out information of all *traced* nodes.

transistors Print out the list of all *transistors*.

watched Print out information of all *watched* nodes.

If no argument is given, the default argument is **watched**.

quit

Exit Msim and return to system. Same as the **exit** command.

reset

Delete all the information about the current circuit. Ready to read in another circuit descriptions.

run [# *steps*]

Simulate the circuit #*steps* steps. If no argument is given, the simulation will continue until the circuit stabilize.

source *file*

Read and execute commands from file *file*.

step [#*steps*]

Simulate the circuit #*steps* steps. Default is 1 step.

trace $node_1$ $node_2$ ...

Trace the list of nodes $node_1$ $node_2$ ... for changes. Report logic value changes as soon as it occurs.

unclock $node_1$ $node_2$ ...

Undefine nodes $node_1$ $node_2$ ... as clocks.

untrace $node_1$ $node_2$ ...

Delete nodes $node_1$ $node_2$ ... from the list of nodes been traced.

unwatch $node_1$ $node_2$ ...

Delete nodes $node_1$ $node_2$ ... from the list of nodes been watched.

watch $node_1$ $node_2$ ...

Watch for the list of nodes $node_1$ $node_2$ .... Print out information about these nodes everytime the simulation stops.

x *node*$_1$ *node*$_2$ ...

Set a list of nodes, *node*$_1$ *node*$_2$ ..., to logic value X, with strength κ_1, which is the weakest among storage nodes.

! *cmd*

Escape to operating system temporarily. Excute the command *cmd* under the operating system environment.

Appendix B

Definitions of Binary Operators

The following are definitions of the binary operators used by our automatic placement system. They are similar to the ones defined in [WoL 87].

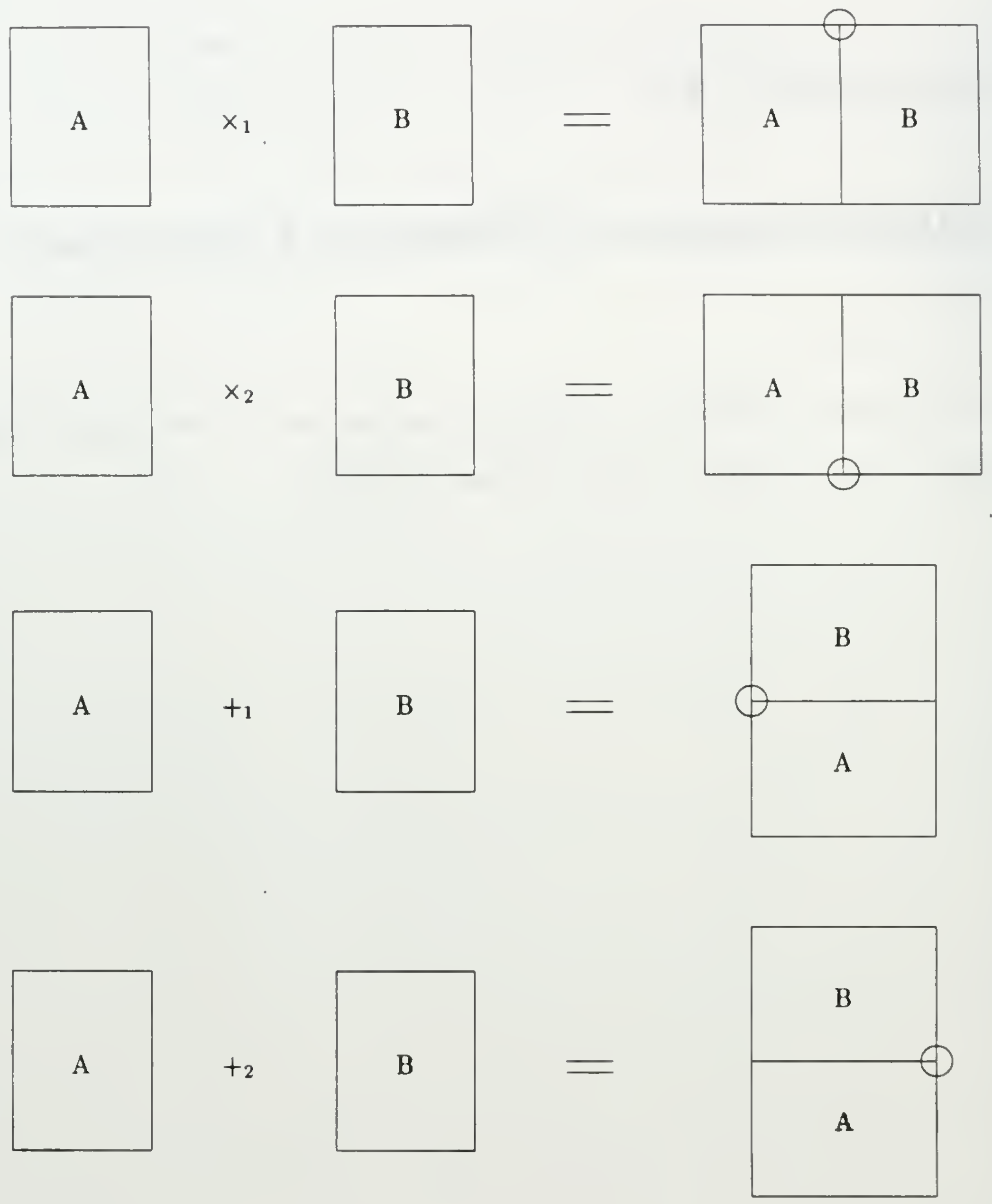


Figure B.1: Orientation of left module = 0, orientation of right module = 0

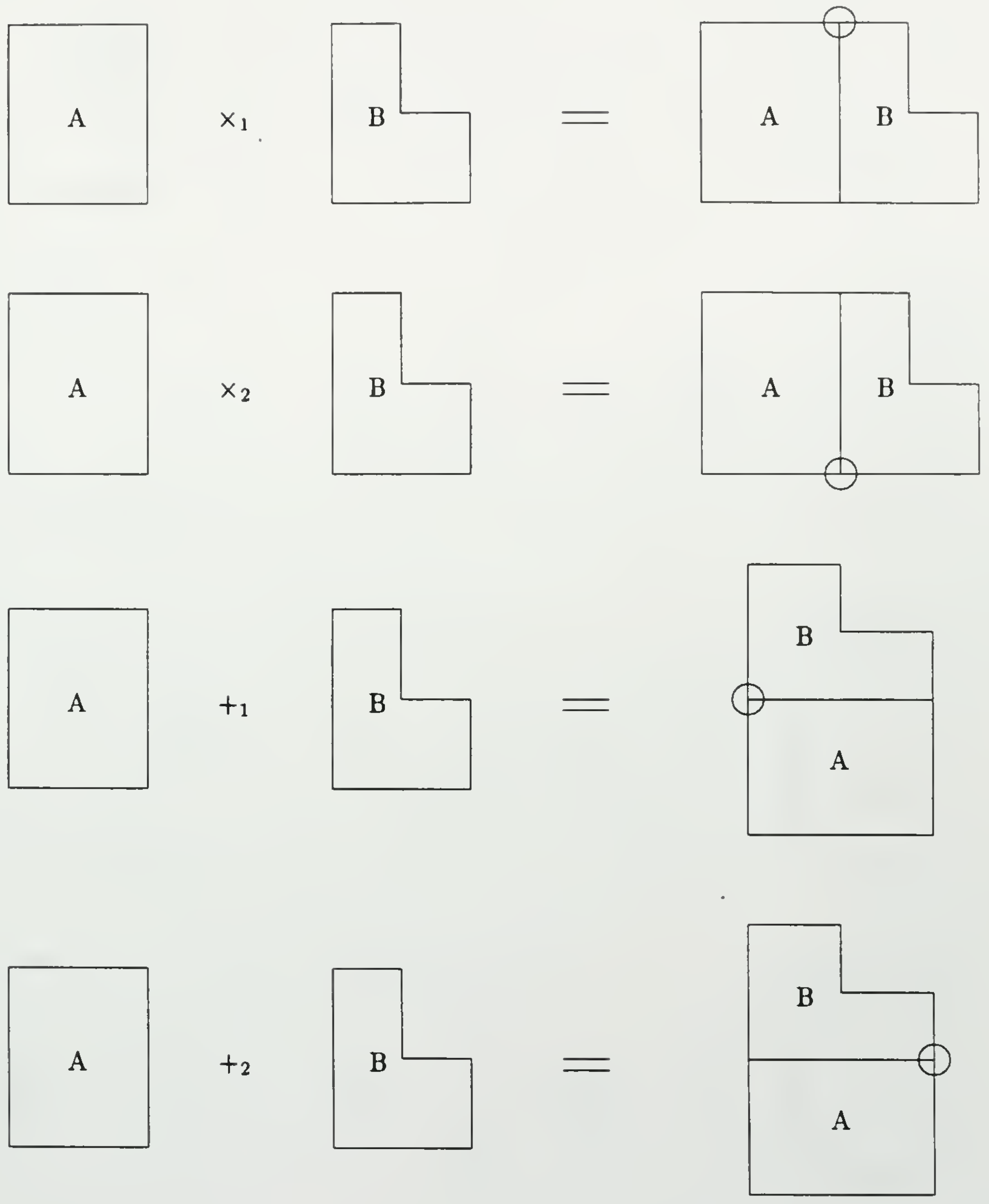


Figure B.2: Orientation of left module = 0, orientation of right module = 1

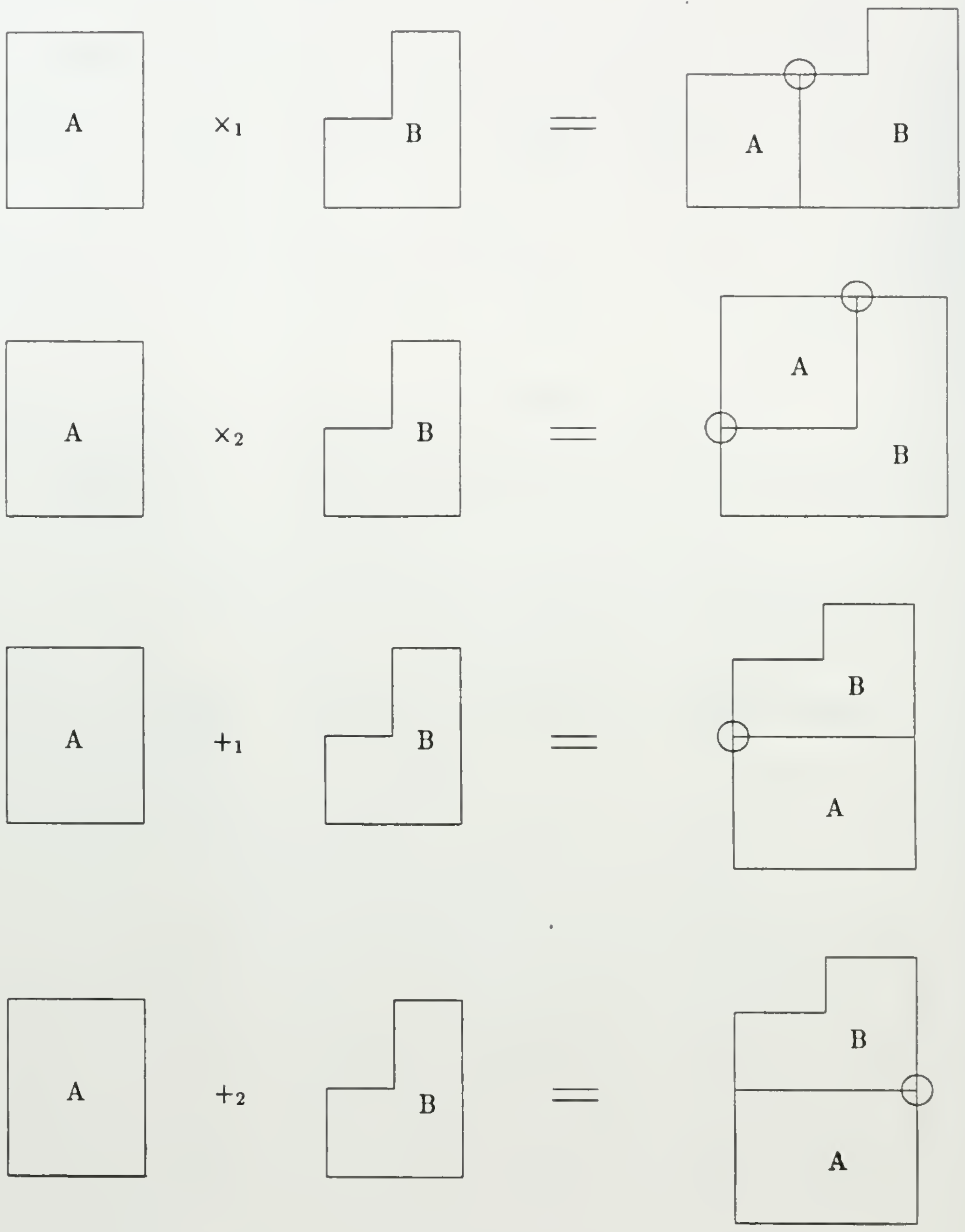


Figure B.3: Orientation of left module = 0, orientation of right module = 2

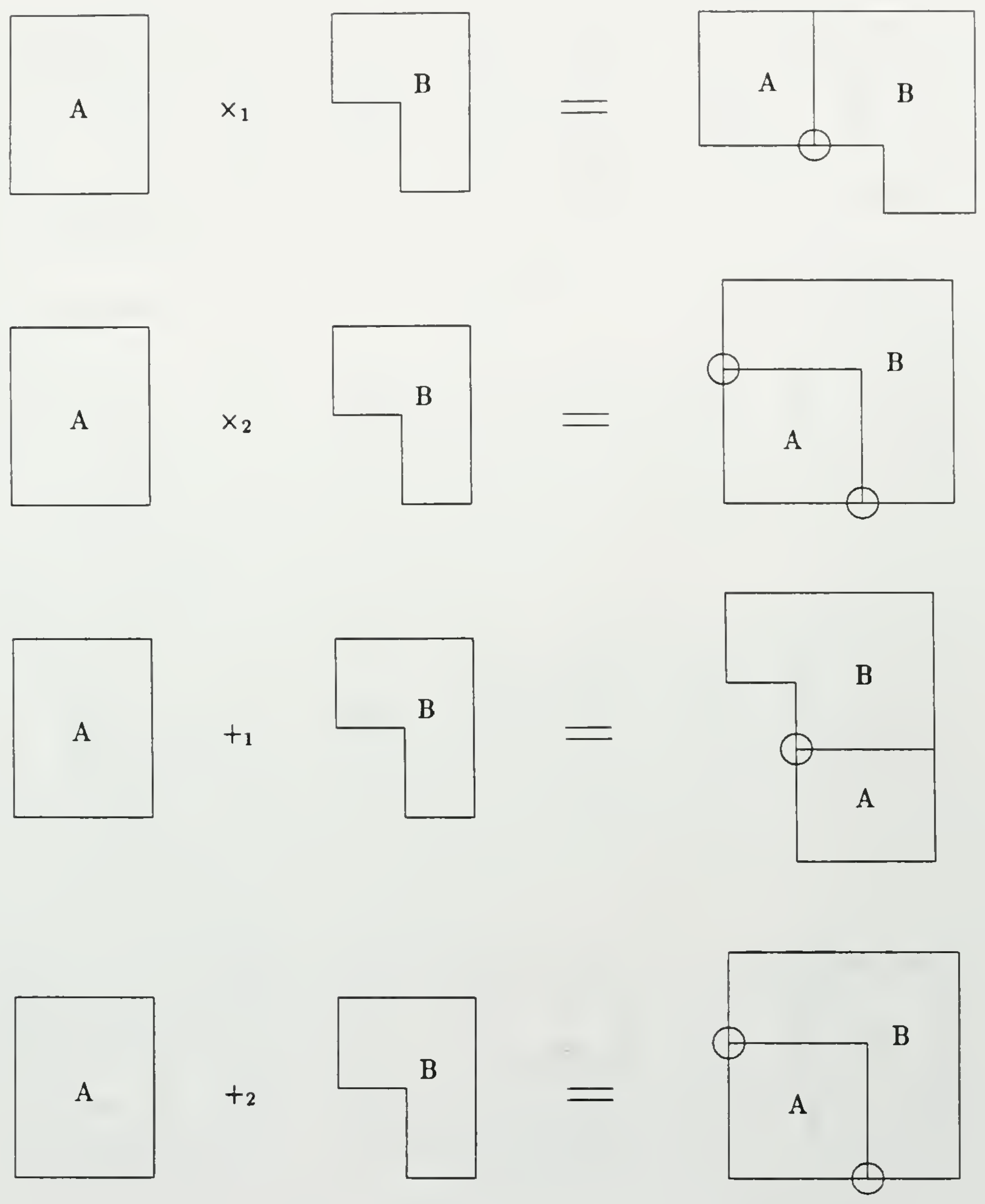


Figure B.4: Orientation of left module = 0, orientation of right module = 3

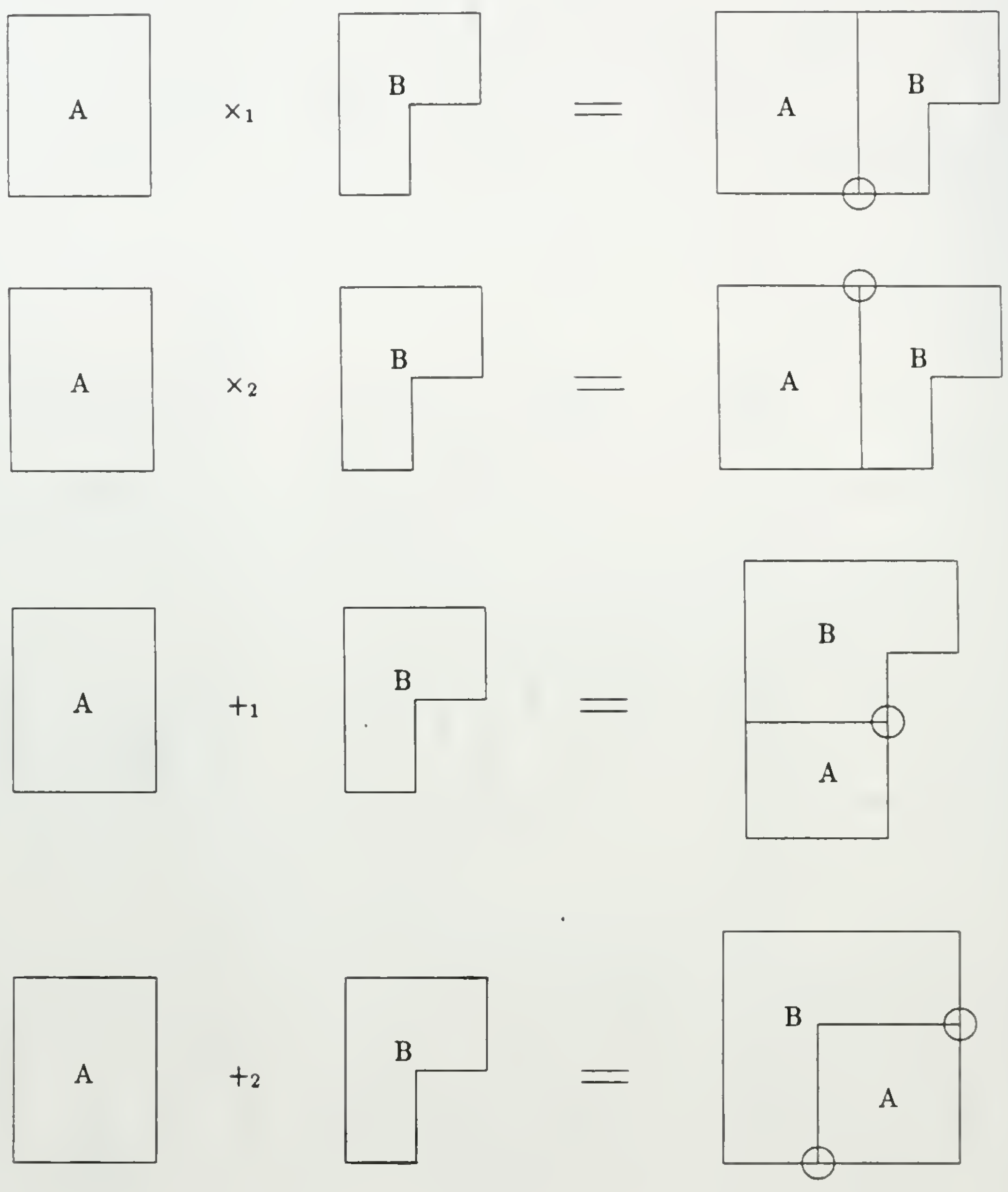


Figure B.5: Orientation of left module = 0, orientation of right module = 4

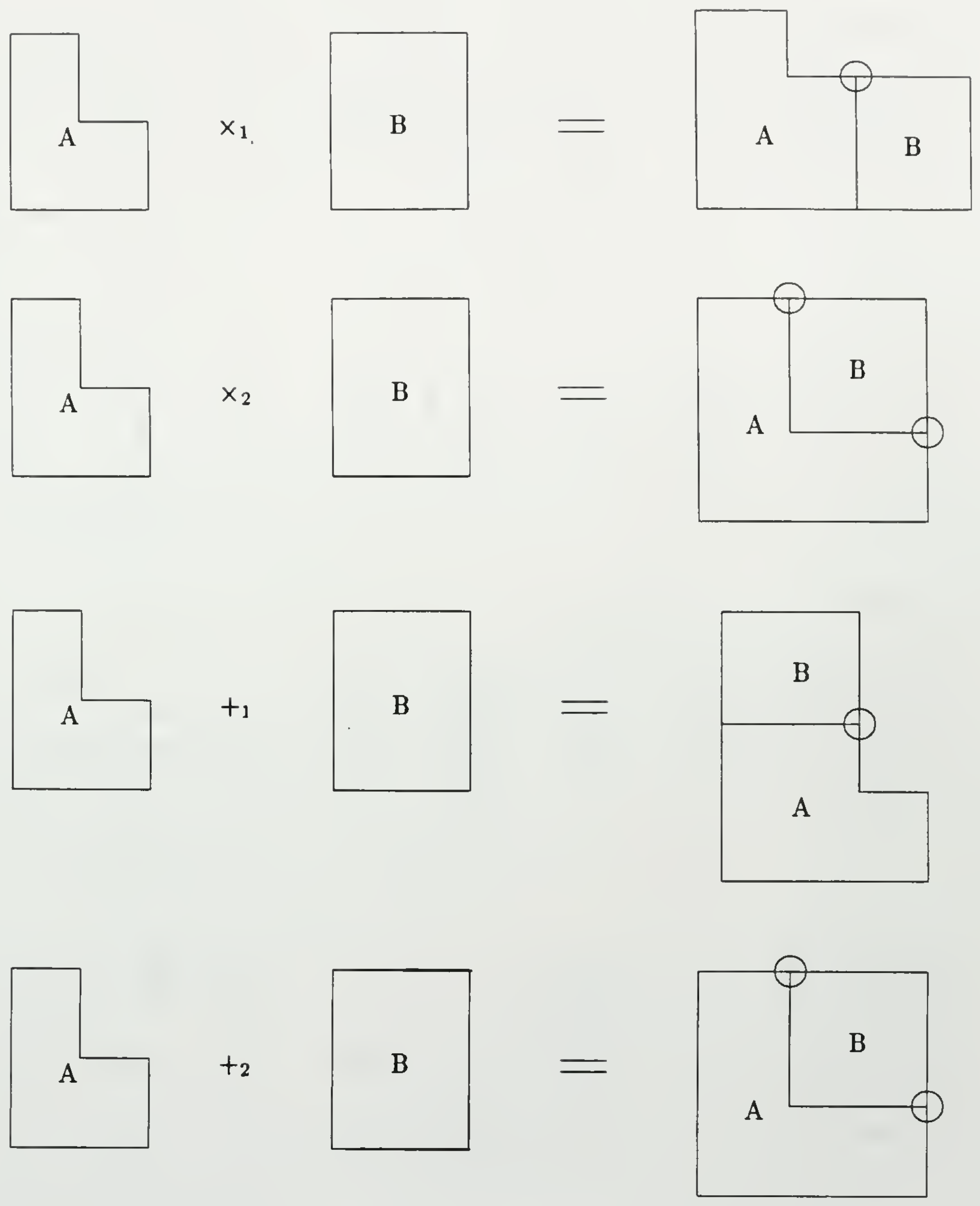


Figure B.6: Orientation of left module = 1, orientation of right module = 0

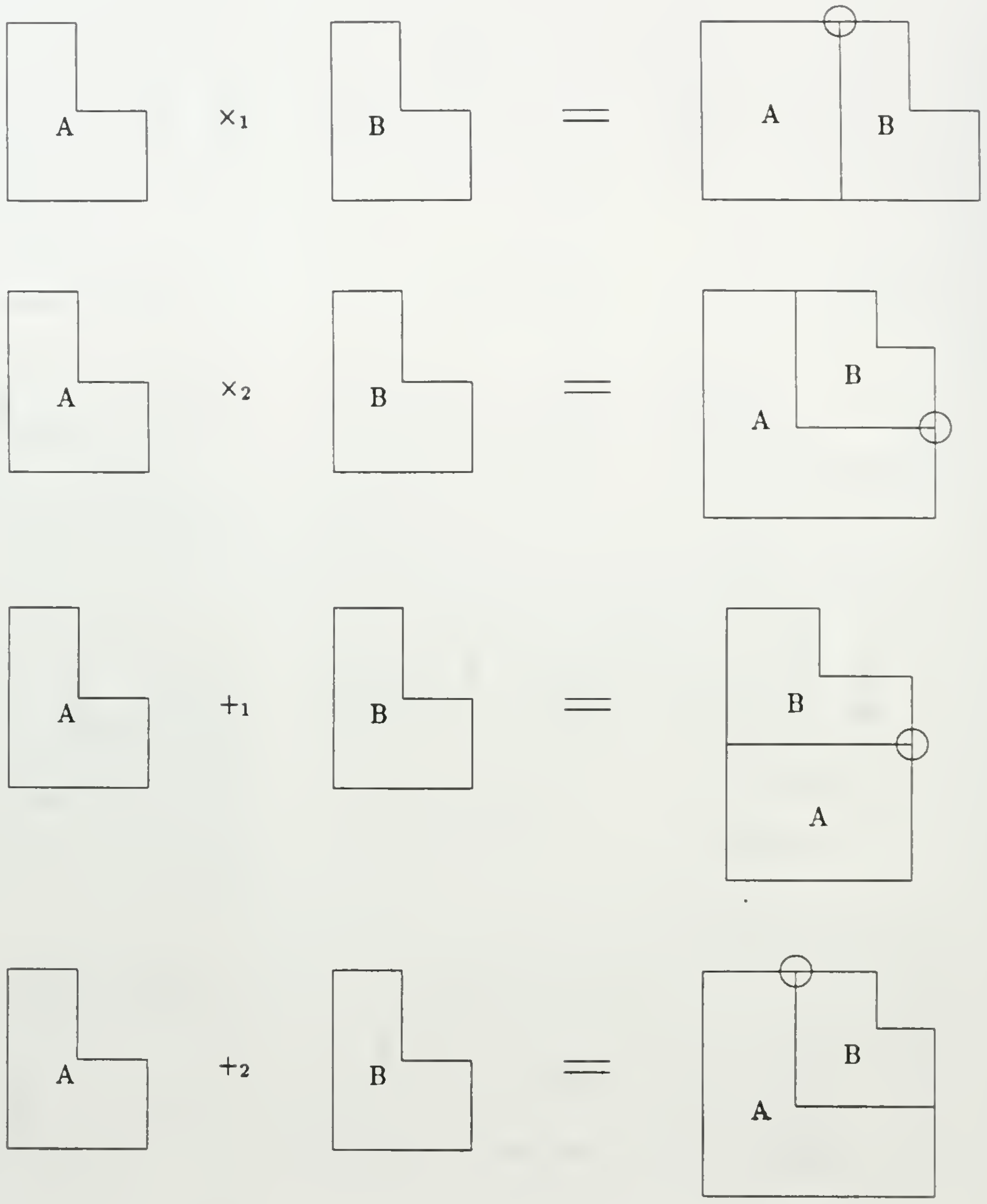


Figure B.7: Orientation of left module = 1, orientation of right module = 1

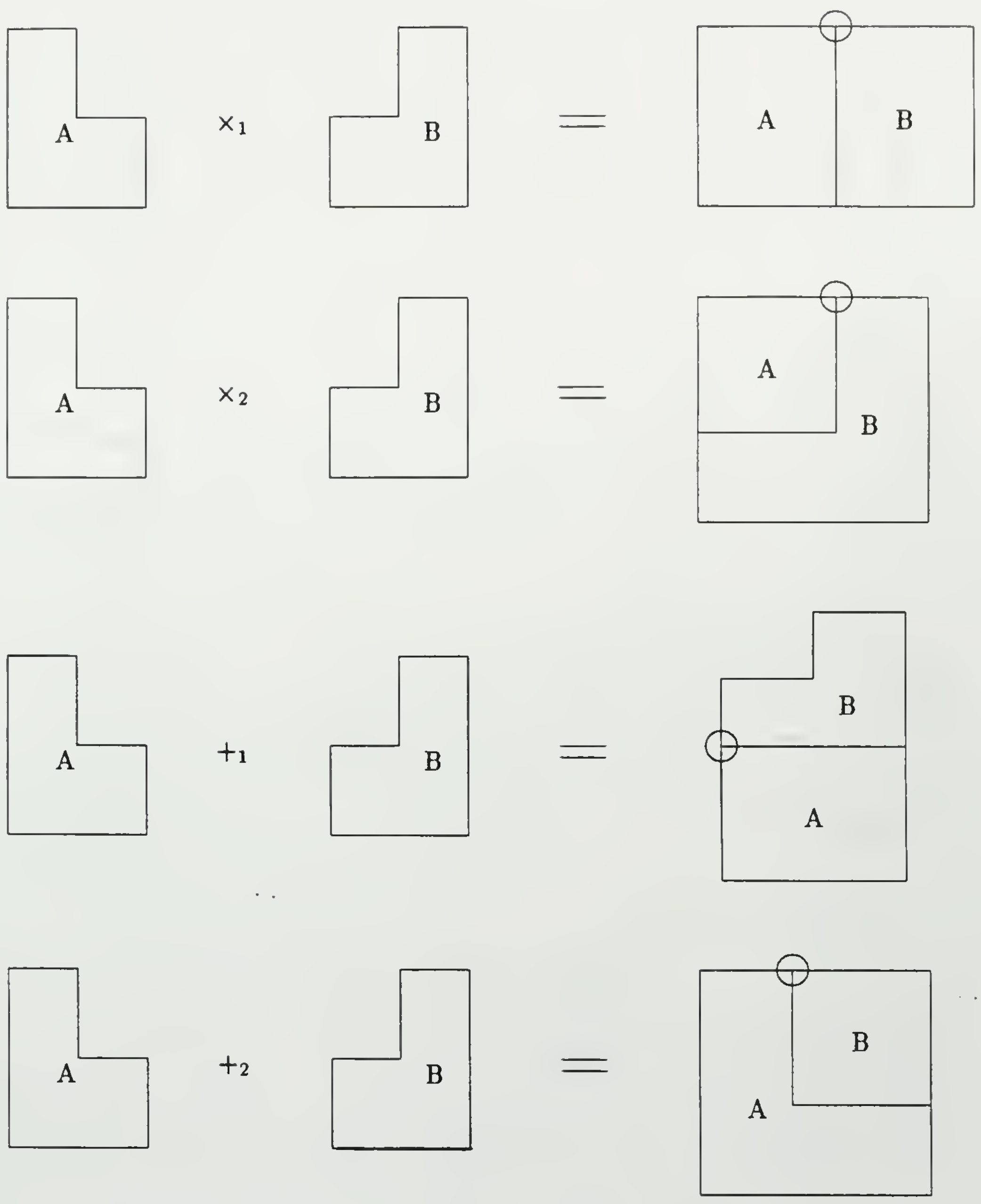


Figure B.8: Orientation of left module = 1, orientation of right module = 2

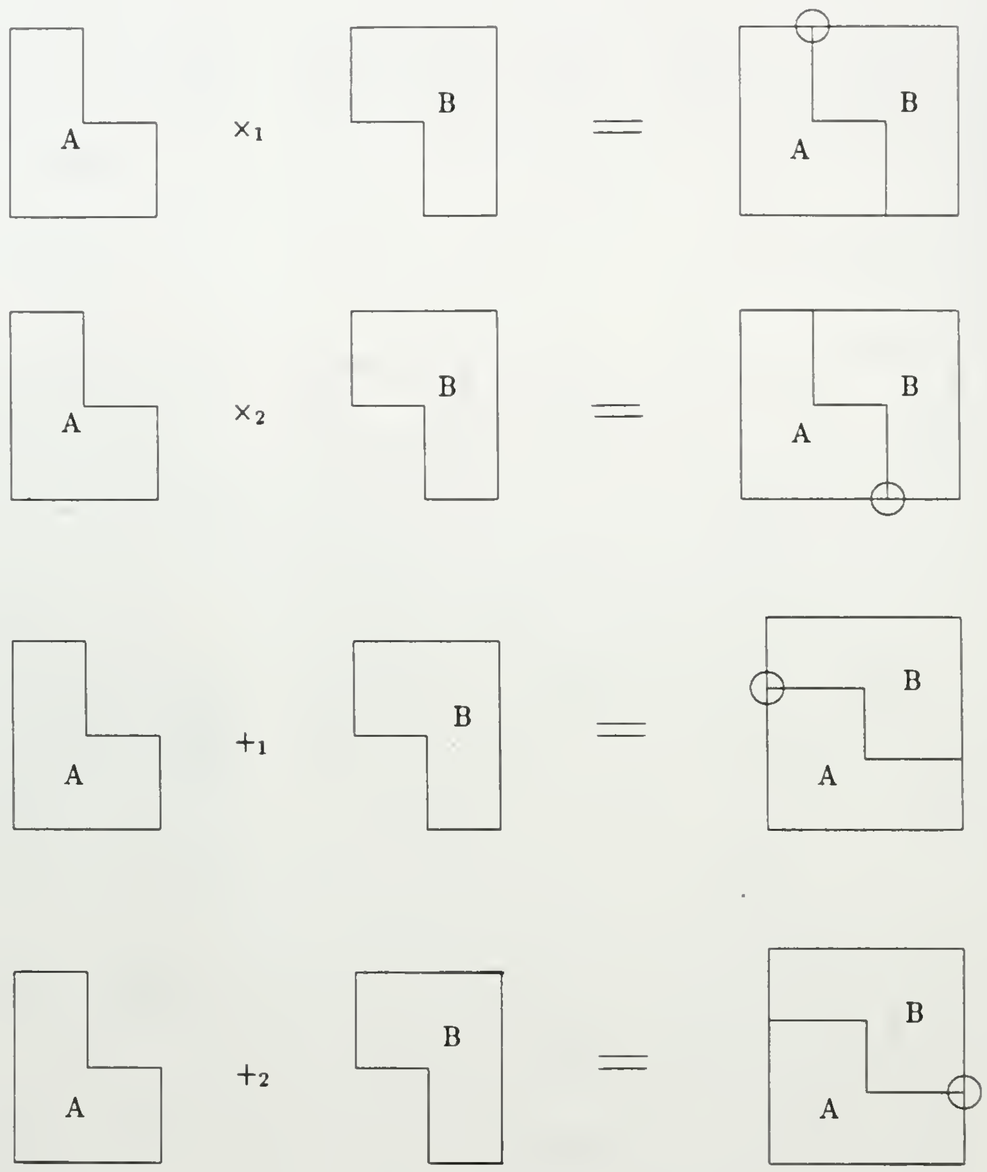


Figure B.9: Orientation of left module = 1, orientation of right module = 3

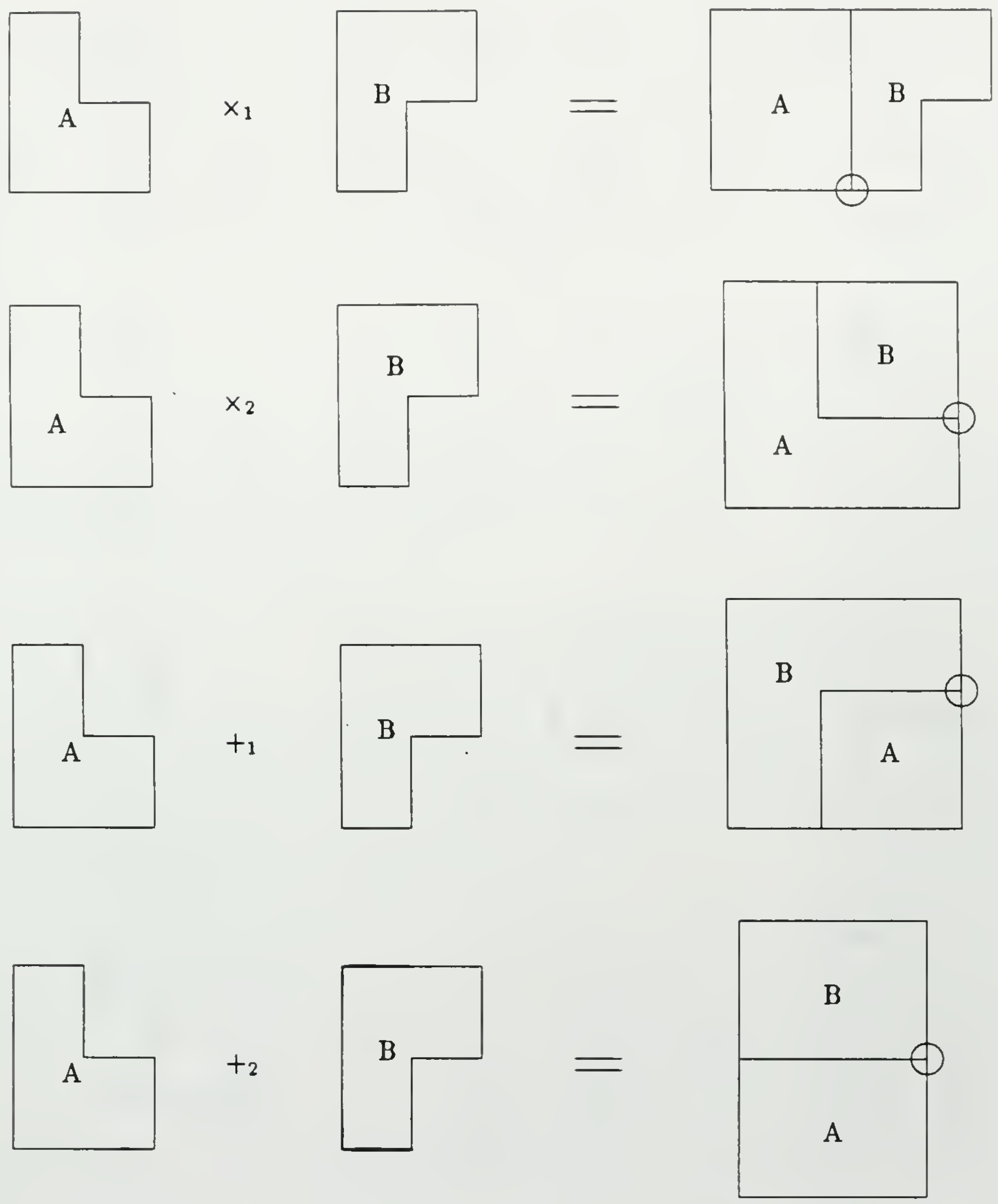


Figure B.10: Orientation of left module = 1, orientation of right module = 4

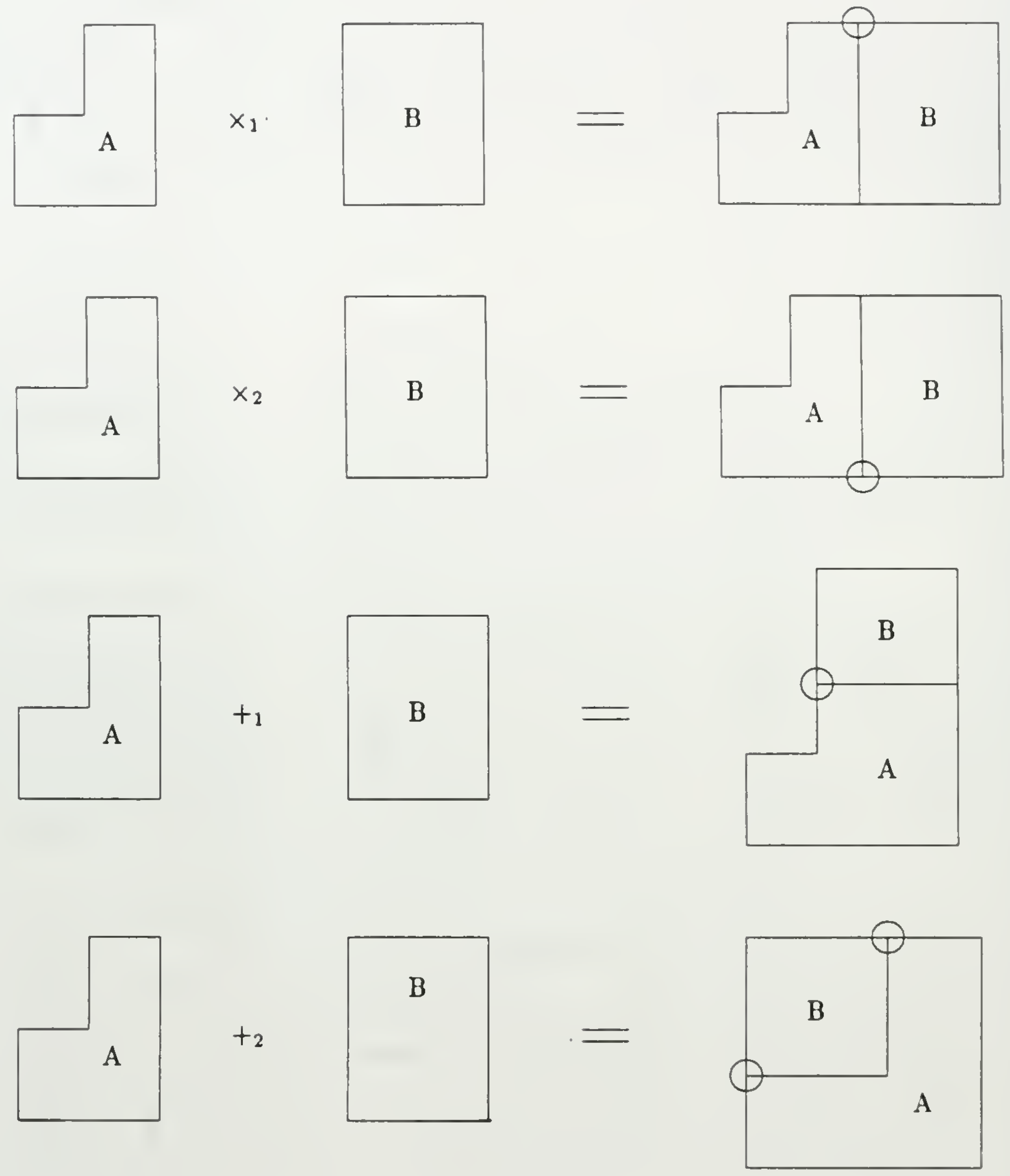


Figure B.11: Orientation of left module = 2, orientation of right module = 0

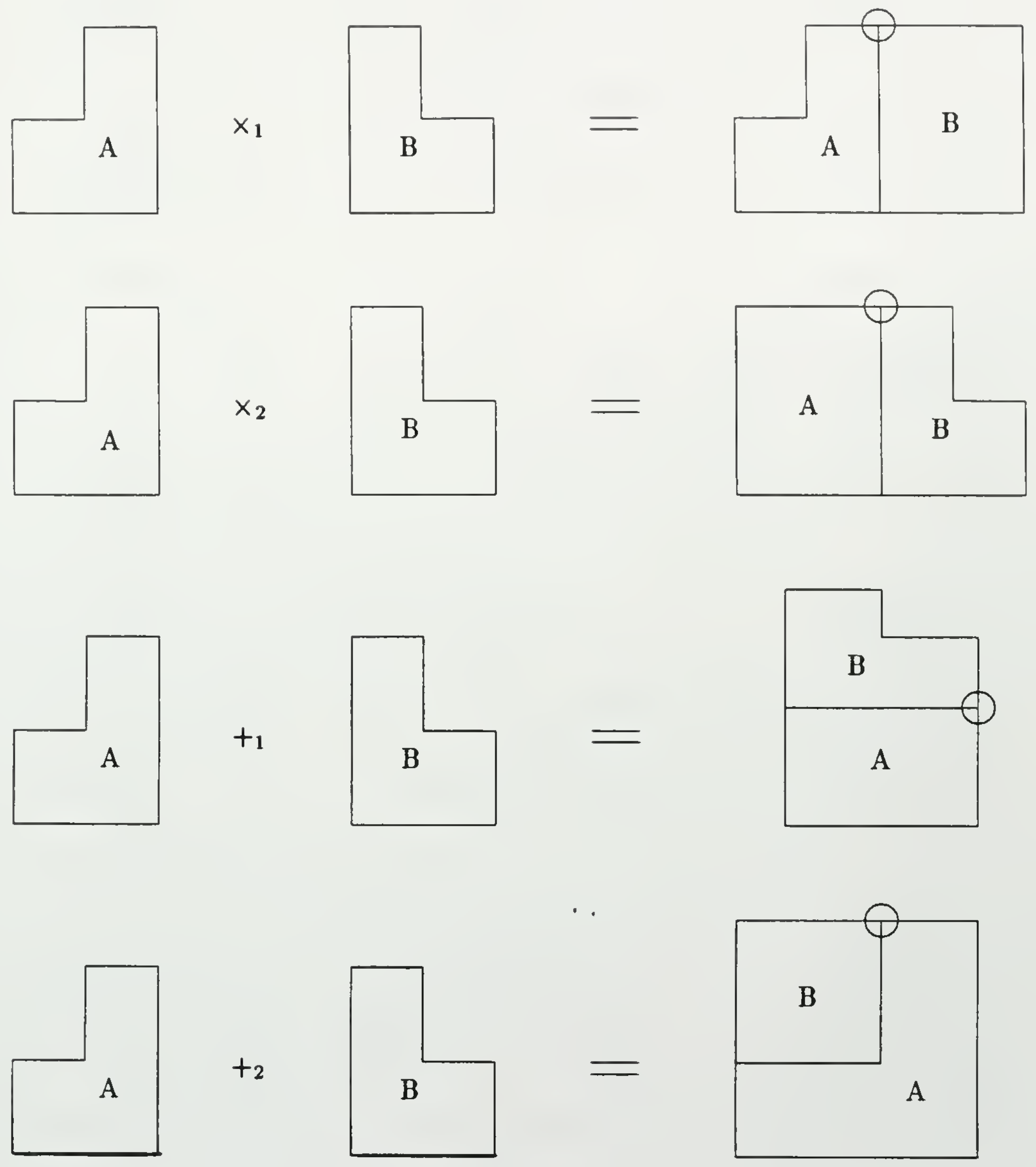


Figure B.12: Orientation of left module = 2, orientation of right module = 1

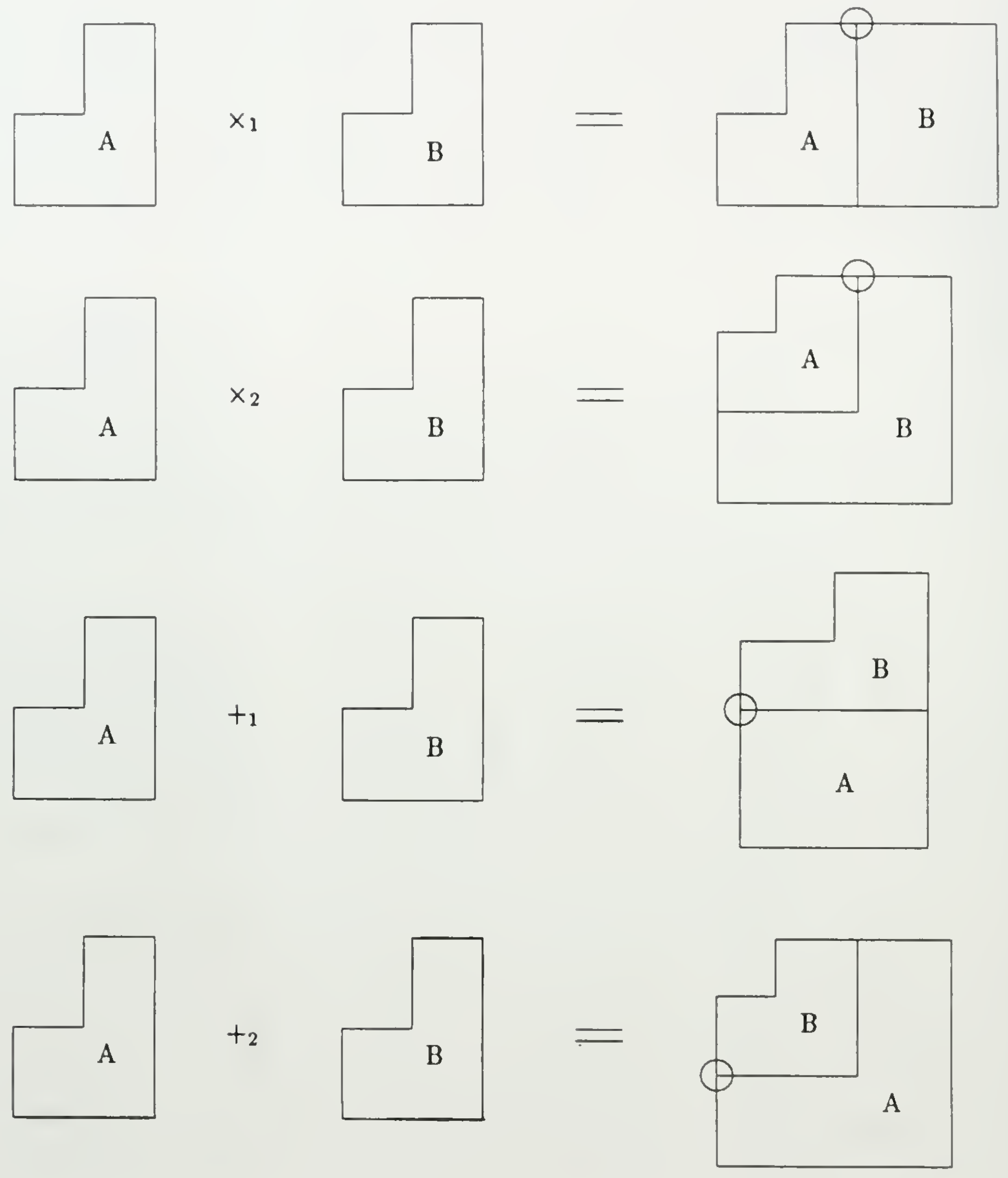


Figure B.13: Orientation of left module = 2, orientation of right module = 2

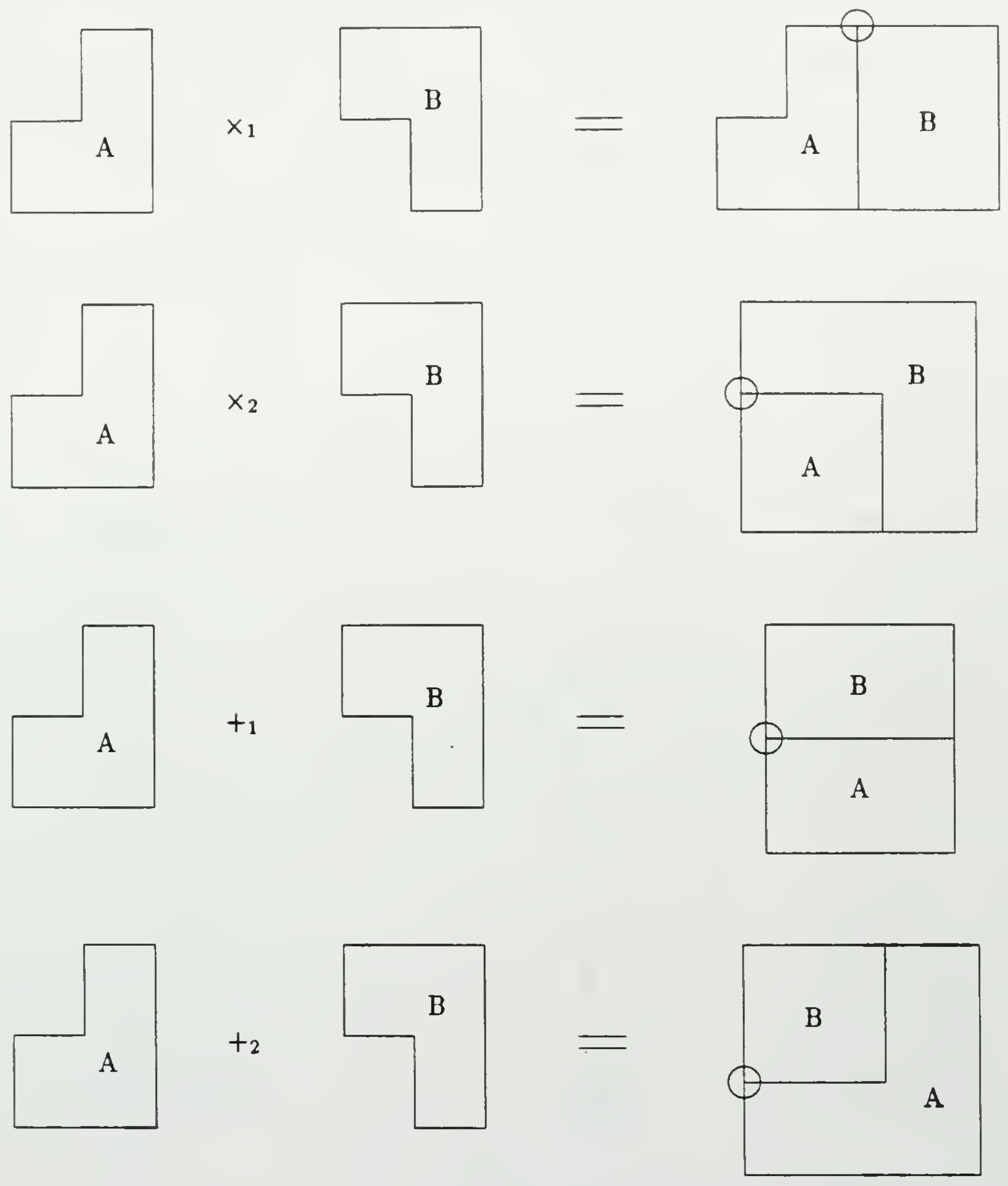


Figure B.14: Orientation of left module = 2, orientation of right module = 3

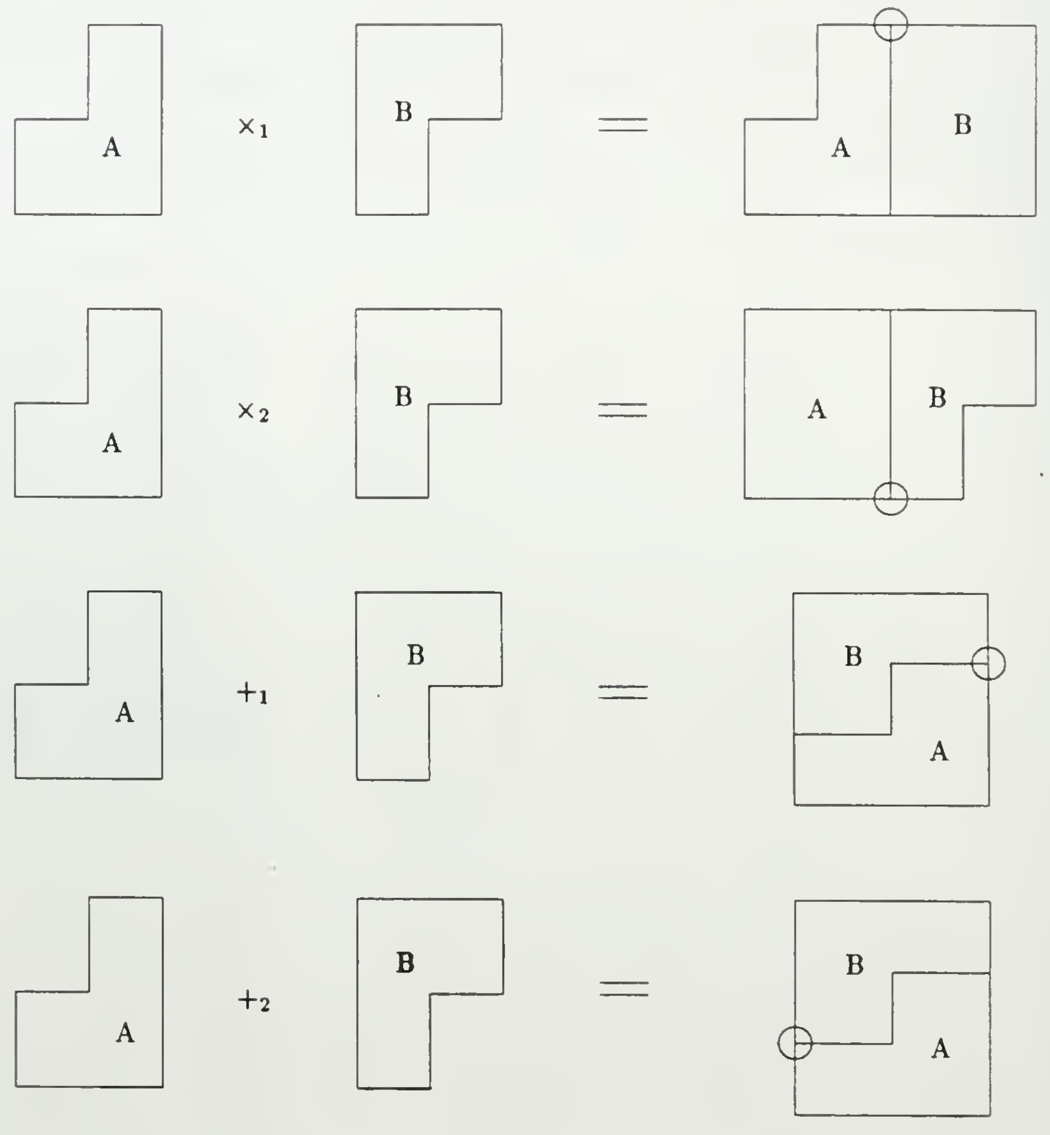


Figure B.15: Orientation of left module = 2, orientation of right module = 4

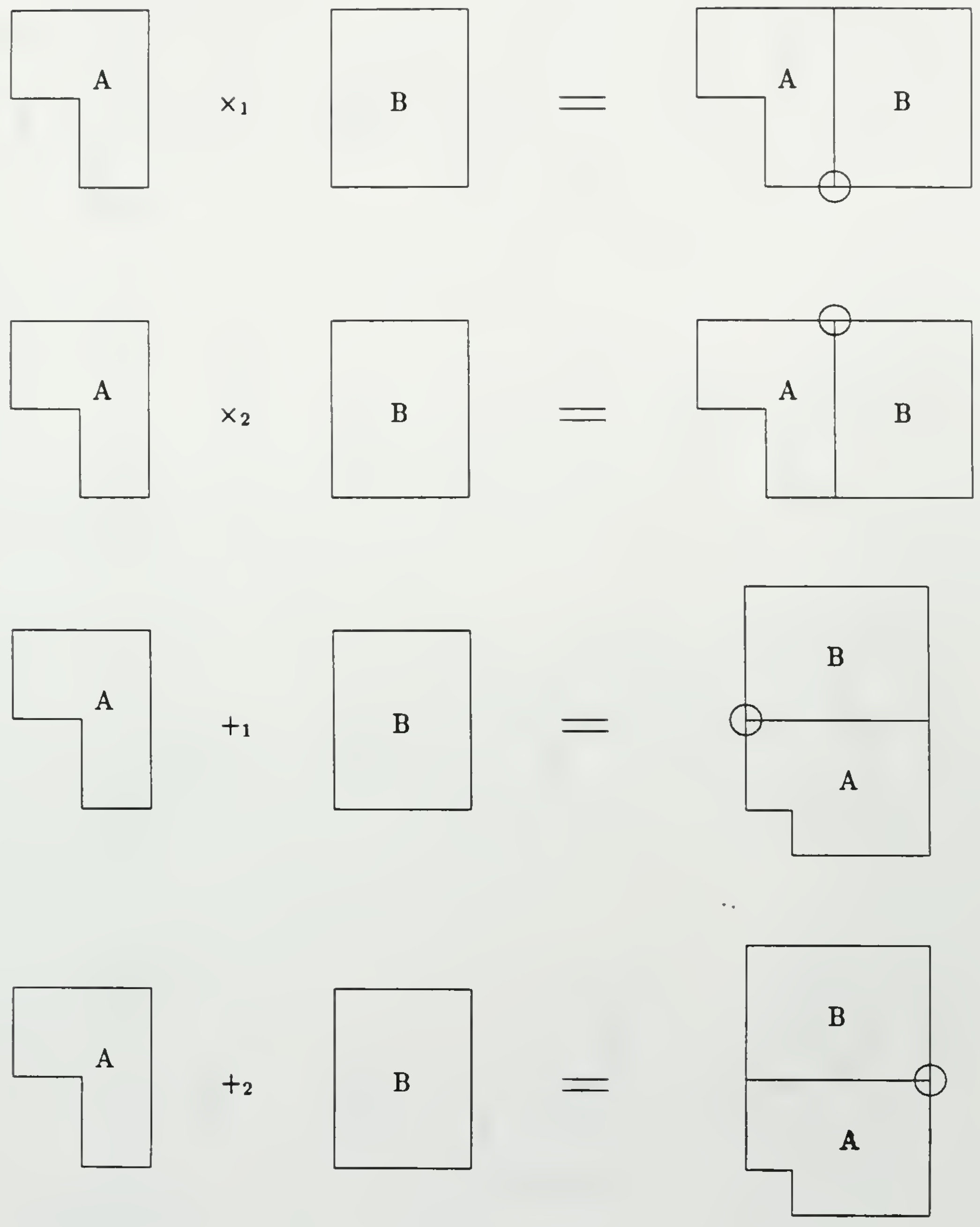


Figure B.16: Orientation of left module = 3, orientation of right module = 0

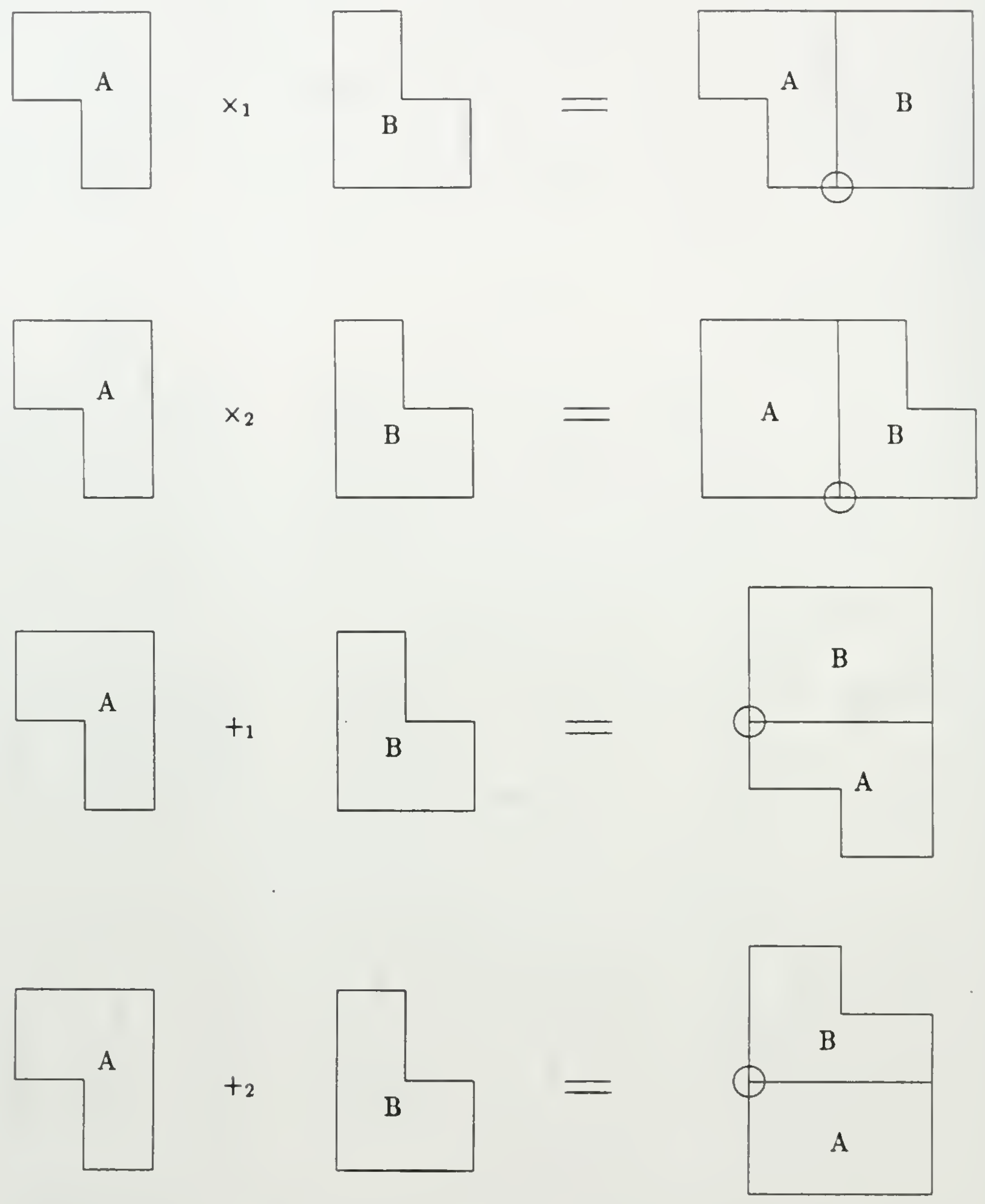


Figure B.17: Orientation of left module = 3, orientation of right module = 1

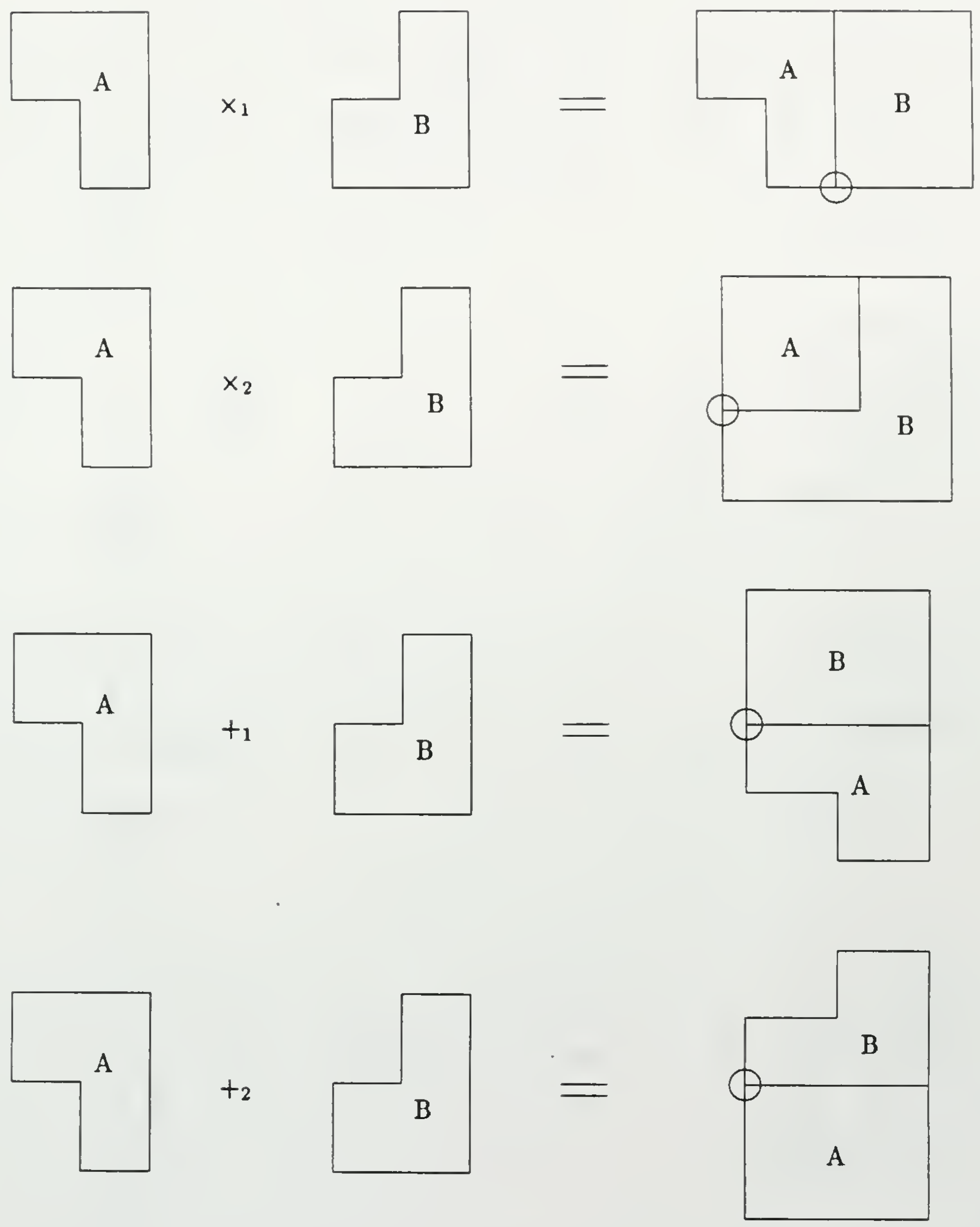


Figure B.18: Orientation of left module = 3, orientation of right module = 2

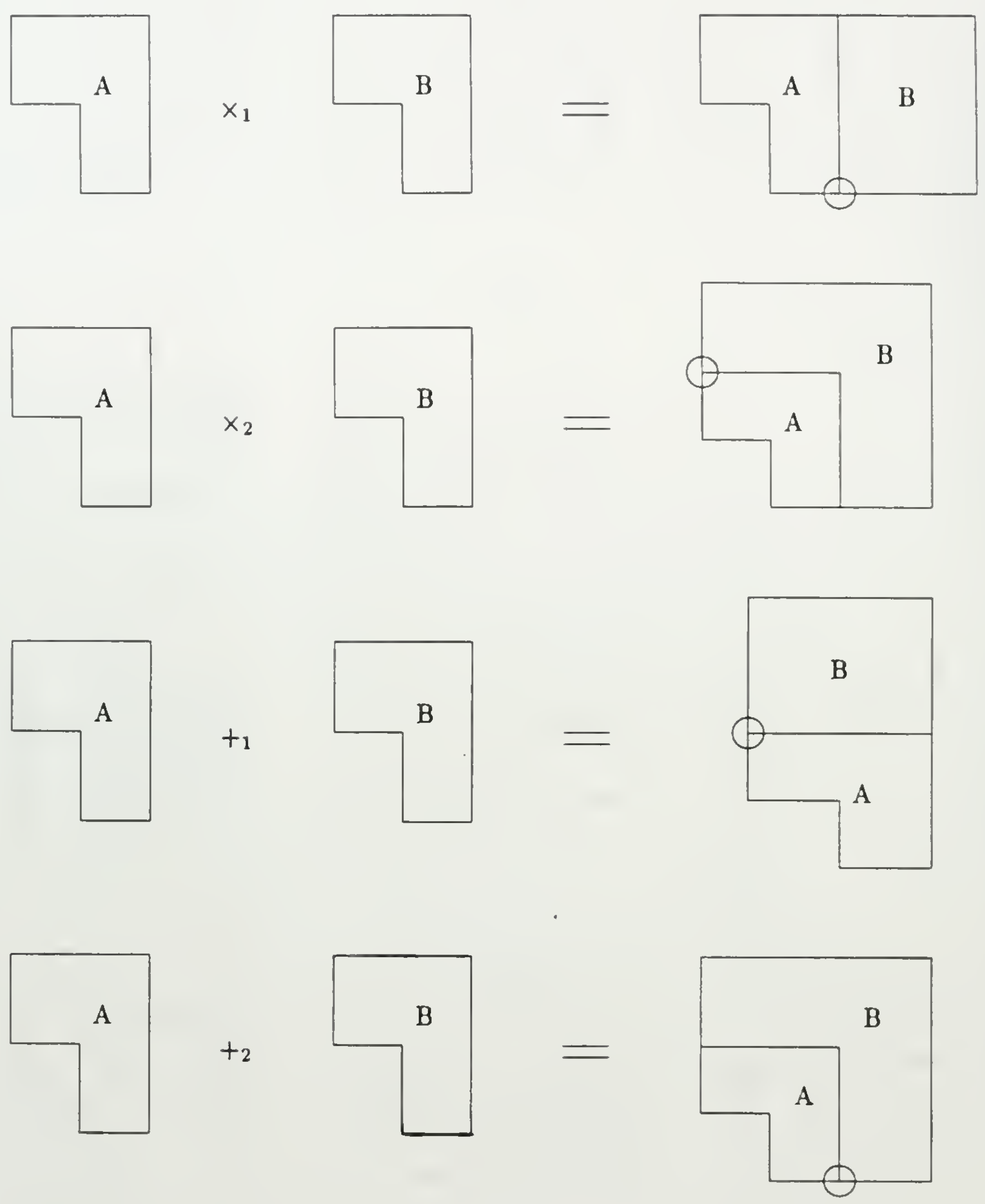


Figure B.19: Orientation of left module = 3, orientation of right module = 3

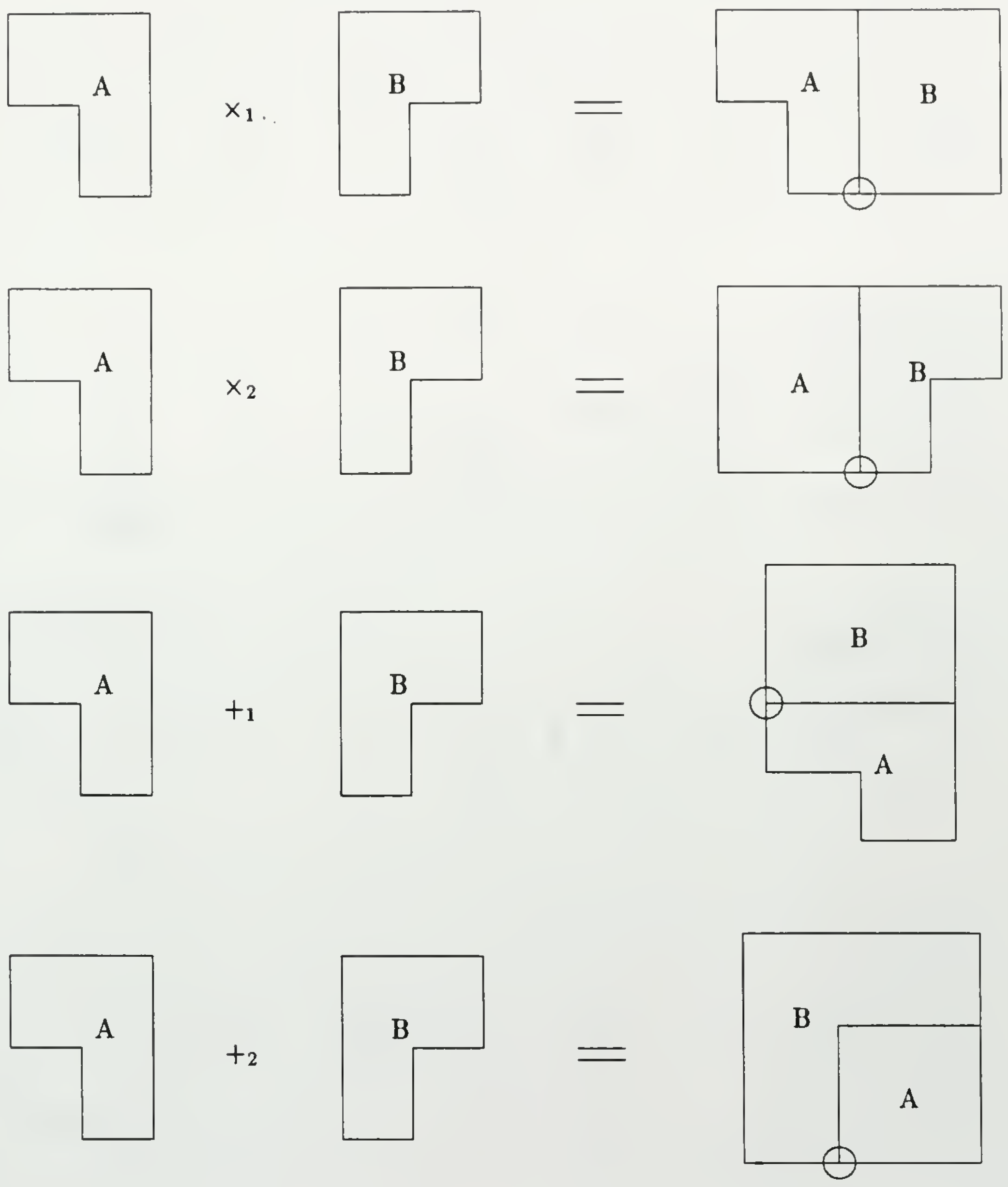


Figure B.20: Orientation of left module = 3, orientation of right module = 4

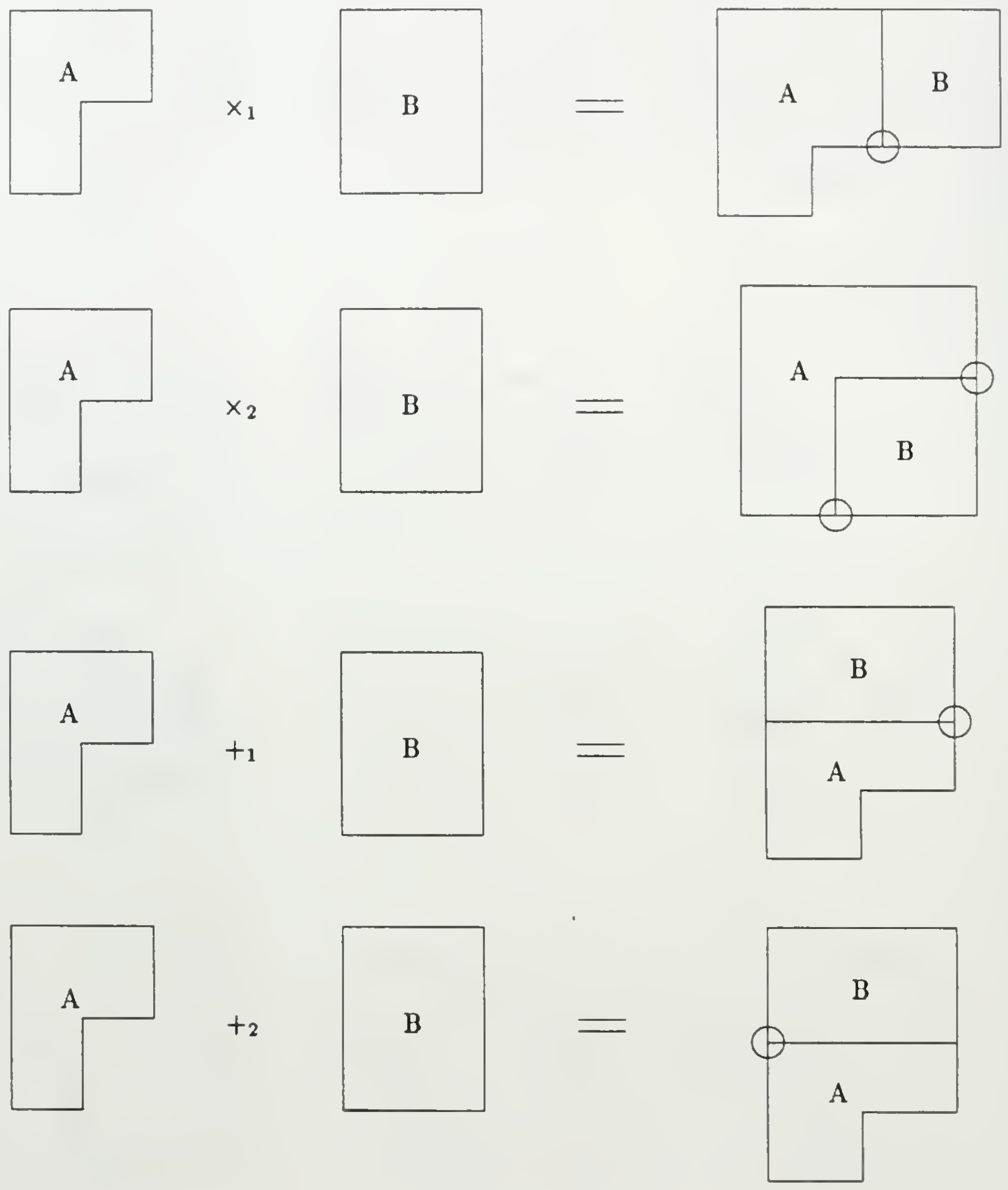


Figure B.21: Orientation of left module = 4, orientation of right module = 0

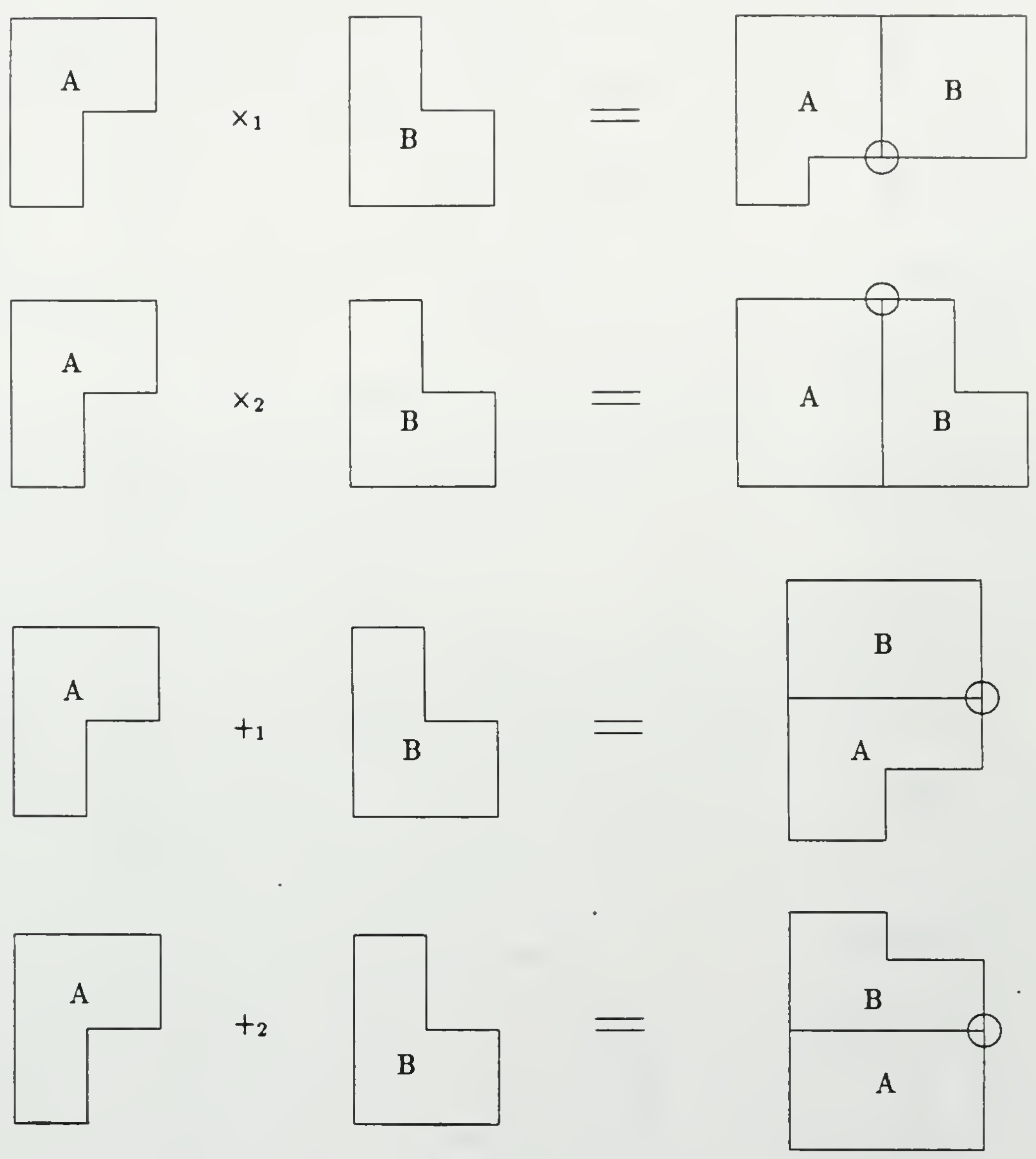


Figure B.22: Orientation of left module = 4, orientation of right module = 1

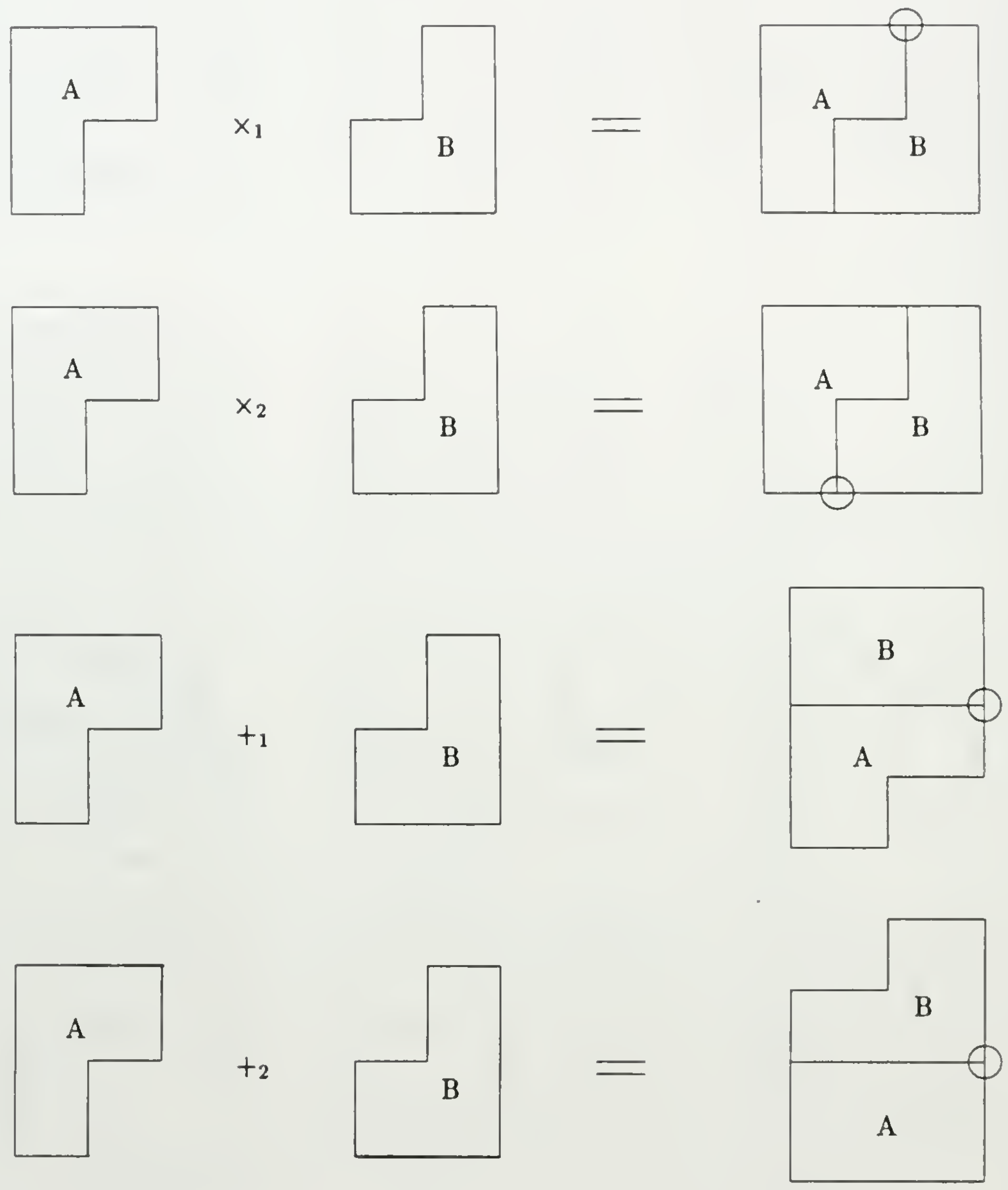


Figure B.23: Orientation of left module = 4, orientation of right module = 2

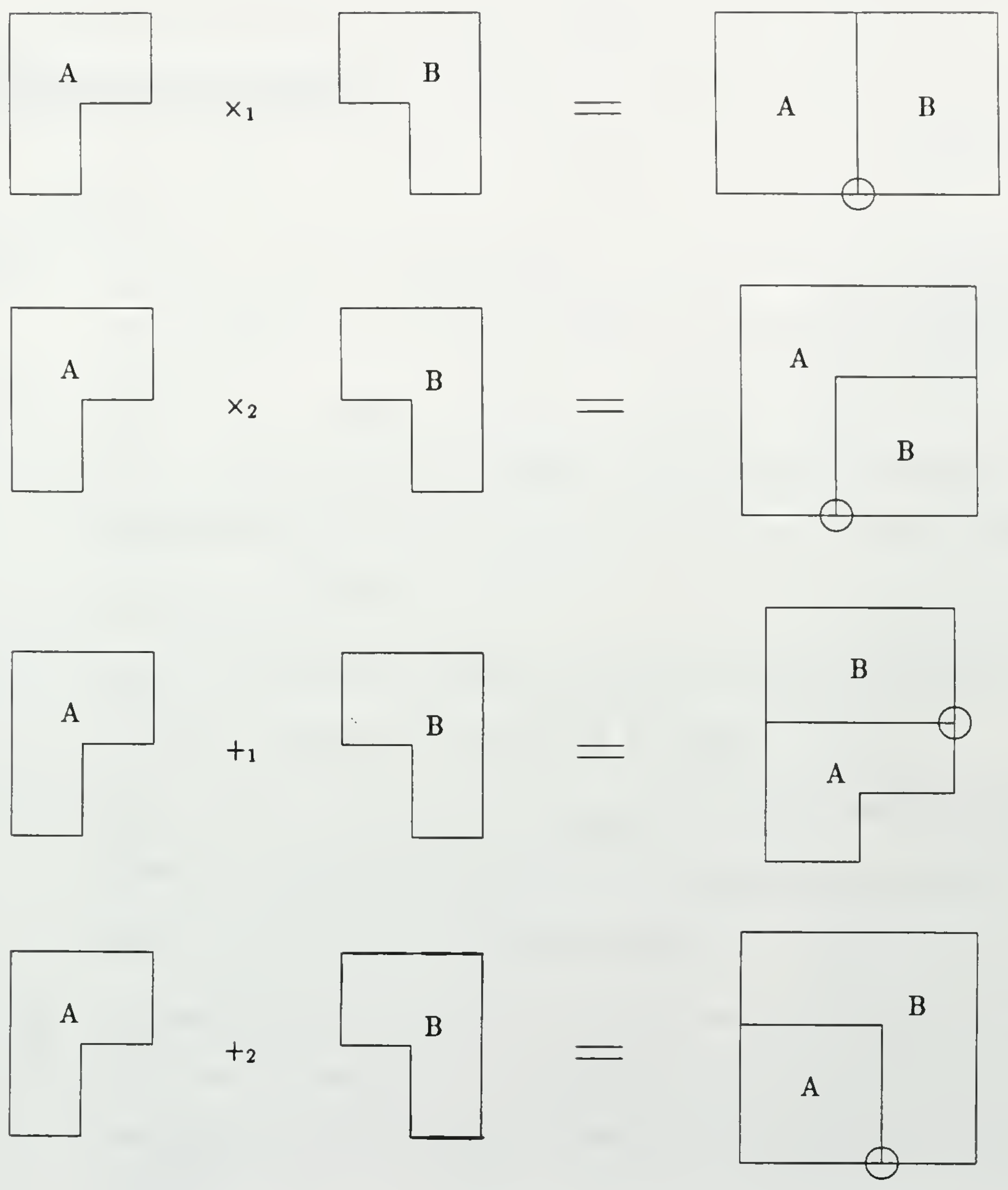


Figure B.24: Orientation of left module = 4, orientation of right module = 3

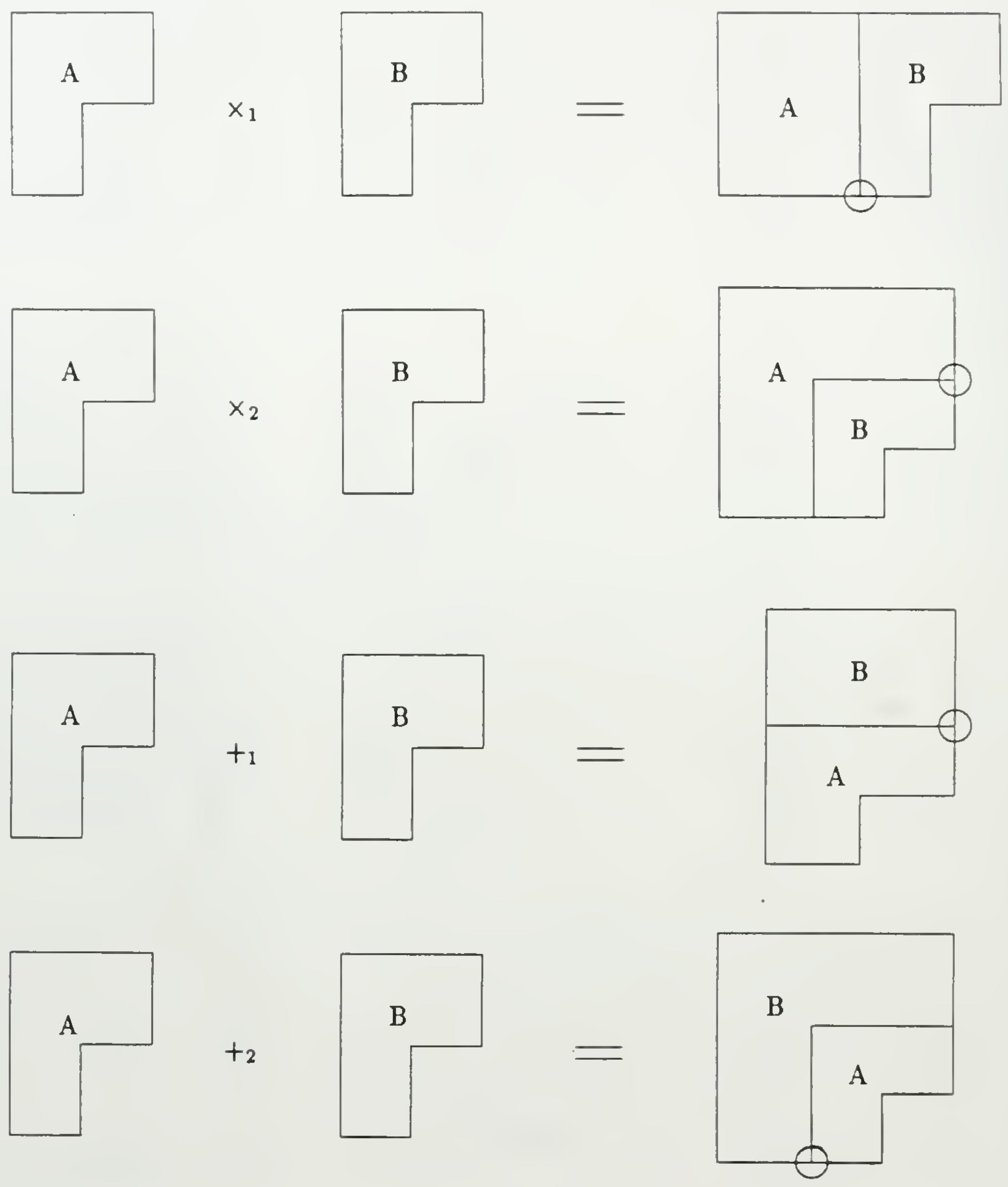


Figure B.25: Orientation of left module = 4, orientation of right module = 4

Bibliography

[Adl 88] Dan Adler, "*A Dynamically-Directed Switch Model for MOS Logic Simulation*," Proceedings of the 25th ACM/IEEE Design Automation Conference, 1988, pp. 506-511.

[ABK 80] V.D. Agrawal, A.K. Bose, P. Kozak, H.N. Nham, and E. Pacas-skewes, "*The Mixed Mode Simulator*," Proceedings of the 17th Design Automation Conference, 1980, pp. 626-633.

[BaT 80] Clark M. Baker and Chris Terman, "*Tools for Verifying Integrated Circuit Designs*," LAMBDA, 4th Quarter, 1980, pp. 22-30.

[BCR 87] Zeev Barzilai, J. Lawrence Carter, Barry K. Rosen, and Joseph D. Rutledge, "*HSS — A High-Speed Simulator*," IEEE Transactions on Computer-Aided Design, Vol. CAD-6, No. 4, July 1987, pp. 601-617.

[BHS 88] Inderpal Bhandari, Mark Hirsch, and Daniel Siewiorek, "*The Min-Cut Shuffle: Toward a Solution for the Global Effect Problem of Min-Cut Placement*," Proceedings of the 25th ACM/IEEE Design Automation Conference, 1988, pp. 681-685.

[Bry 80] Randal E. Bryant, "*An Algorithm for MOS Logic Simulation*," LAMBDA, 4th Quarter, 1980, pp. 46-53.

[Bry 81] Randal E. Bryant, "*MOSSIM: A Switch-Level Simulator for MOS LSI*," Proceedings of the 18th Design Automation Conference, 1981, pp. 786-790.

[Bry 83] Randal E. Bryant, "*Race Detection in MOS Circuits*," VLSI'83, F. Anceau and E.J. Aas, eds, North-Holland, 1983, pp. 85-95.

[Bry 84] Randal E. Bryant, "*A Switch-Level Model and Simulator for MOS Digital Systems*," IEEE Transactions on Computers, Vol. C-33, No. 2, Feb. 1984, pp. 160-177.

[Bry 85] Randal E. Bryant, "*Symbolic Verification of MOS Circuits*," Proceedings of Chapel Hill Conference on VLSI, H. Fuchs, ed., 1985, pp. 419-438.

[Bry 87] Randal E. Bryant, "*A Survey of Switch-Level Algorithms*," IEEE Design & Test, August 1987, pp. 26-40.

[BBB 87] Randal E. Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, and Thomas Sheffler, "*COSMOS: A Compiled Simulator for MOS Circuits*," Proceedings of the 24th ACM/IEEE Design Automation Conference, 1987, pp. 9-16.

[CaS 78] Glenn R. Case and Jerry D. Stauffer, "*Salogs-IV: A Program to Perform Logic Simulation and Fault Diagnosis*," Proceedings of the 15th Design Automation Conference, June 1978, pp. 392-397.

[ChC 78] Robert C. Chen and James E. Coffman, "*Multi-Sim, A Dynamic Multi-Level Simulator*," Proceedings of the 15th Design Automation Conference, 1978, pp. 386-391.

[Den 82] Monty M. Denneau, "*The Yorktown Simulation Engine*," Proceedings of the 19th Design Automation Conference, 1982, pp. 55-59.

[DJF 87] L.-P. Demers, P. Jacques, S. Fauvel, and E. Cerny, "*CHESHIRE: An Object-Oriented Integration of VLSI CAD Tools*," Proceedings of the 24th ACM/IEEE

Design Automation Conference, 1987, pp. 750-756.

[Don 80] W.E. Donath, "*Complexity Theory And Design Automation*," Proceedings of the 17th Design Automation Conference, 1980, pp. 412-419.

[For 87] Rob Forbes, "*Heuristic Acceleration of Force-directed Placement*," Proceedings of the 24th ACM/IEEE Design Automation Conference, 1987, pp. 735-740.

[FrG 86] Patrice Frison and Eric Gautrin, "*MADMACS: A New VLSI Layout Macro Editor*," Proceedings of the 23rd Design Automation Conference, 1986, pp. 654-658.

[Gar 77] Robert I. Gardner, "*Multi-Level Modeling in SARA*," Proceedings of the Symposium on Design Automation and Microprocessors, Palo Alto, Feb. 1977, pp. 63-67.

[GaJ 79] Michael R. Garey and David S. Johnson, *Computers And Intractability*, W.H. Freeman And Company, New York, 1979.

[ClD 85] Lance A. Glasser and Daniel W. Dobberphul, *The Design And Analysis of VLSI Circuits*, Addison-Wesley, 1985.

[GrS 84] J.W. Greene and K.J. Supowit, "*Simulated Annealing Without Rejected Moves*," Proceedings of the International Conference on Computer Design, 1984, pp. 658-663.

[Gro 87] Lov K. Grover, "*Standard Cell Placement Using Simulated Sintering*," Proceedings of the 24th ACM/IEEE Design Automation Conference, 1987, pp. 56-59.

[HaO 84] Gordon T. Hamachi and John K. Ousterhout, "*A Switchbox Router with Obstacle Avoidance*," Proceedings of the 21st ACM/IEEE Design Automation Conference, 1984, pp. 173-179.

[HiC 87] Dwight D. Hill and David R. Coelho, *Multi-level Simulation for VLSI Design*, Kluwer Academic Publishers, 1987.

[Hiv 79] Dwight Hill and Willem vanCleemput, "*SABLE: A Tool for Generating Structured, Multi-Level Simulations*," Proceedings of the 16th Design Automation Conference, 1979, pp. 272-279.

[Hor 85] E. Horbst, ed., *VLSI 85 – VLSI Design of Digital Systems*, North-Holland, 1985.

[HWA 76] M. Hanan, P.K. Wolff, and B.J. Agule, "*Some Experimental Results on Placement Techniques*," Proceedings of the 13th Design Automation Conference, 1976, pp. 214-224.

[Kat 82] Randy H. Katz, "*A Database Approach for Managing VLSI Design Data*," Proceedings of the 19th Design Automation Conference, 1982, pp. 274-282.

[KlB 87] Ralph-Michael Kling and Prithviraj Banerjee, "*ESP: A New Standard Cell Placement Package Using Simulated Evolution*," Proceedings of the 24th ACM/IEEE Design Automation Conference, 1987, pp. 60-66.

[KrP 82] E. Kronstadt and G. Pfister, "*Software Support for the Yorktown Simulation Engine*," Proceedings of the 19th Design Automation Conference, 1979, pp. 60-64.

[LaK 85] Richard H. Lathrop and Robert S. Kirk, "*An Extensible Object-Oriented Mixed-Mode Functional Simulation System*," Proceedings of the 22nd ACM/IEEE Design Automation Conference, 1985, pp. 630-636.

[Lau 80] Ulrich Lauther, "*A Min-Cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation*," Proceedings of the 16th Design Automation Conference, 1979, pp. 1-4.

[LeS 82] E. Lelarasmee and A. Sangiovanni-Vincentelli, "*RELAX: A New Circuit Simulator for Large Scale MOS Integrated Circuits*," Proceedings of the 19th ACM/IEEE Design Automation Conference, 1982, pp. 682-691.

[LMS 86] Roger Lipsett, Erich Marschner, and Moe Shahdad, "*VHDL – The Language*," IEEE Design & Test, April 1986.

[Loo 79] K.J. Loosemore, "*Automated Layout of Integrated Circuits*," Proceedings of the IEEE International Symposium on Circuits and Systems, 1979, pp. 665-668.

[MaG 88] Sivanarayana Mallela and Lov K. Grover, "*Clustering Based Simulated Annealing for Standard Cell Placement*," Proceedings of the 25th ACM/IEEE Design Automation Conference, 1988, pp. 312-317.

[Man 82] M. Morris Mano, *Computer System Architecture*, Second Edition, Prentice-Hall, 1982.

[MIB 86] Bill McCalla, Beatriz Infante, Dennis Brzezinski, and Joe Beyers, "*Chip-Buster VLSI Design System*," Proceedings of IEEE International Conference on Computer-Aided Design, 1986, pp. 20-23.

[MeC 80] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[MGS 89] Julia Miller, Klaus Groning, Gerhard Schulz, and Charles White, "*The Object-Oriented Integration Methodology of the Cadlab Work Station Design Environment*," Proceedings of the 26th ACM/IEEE Design Automation Conference, 1989, pp. 807-810.

[MWL 87] Thomas R. Mueller, D.F. Wong, and C.L. Liu, "*An Enhanced Bottom-Up Algorithm for Floorplan Design*," 1987, pp. 524-527.

[Nag 75] L. S. Nagel, "*SPICE2: A Computer Program to Simulate Semiconductor Circuits*," ERL Memo ERL-M520, University of California, Berkeley, May 1975.

[OrR 83] G. M. Ordy and C. W. Rose, "*The N.2 System*," Proceedings of the 20th Design Automation Conference, 1983, pp. 520-526.

[Ous 84] John K. Ousterhout, "*Switch-Level Delay Models for Digital MOS VLSI*," Proceedings of the 21st Design Automation Conference, 1984, pp. 542-548.

[ONC 86] Peter Odryna, Kevin Nazareth, and Carl Christensen, "*A Workstation-Based Mixed Mode Circuit Simulator*," Proceedings of the 23rd Design Automation Conference, 1986, pp. 186-192.

[OHM 84] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott and G. S. Taylor, "*Magic: A VLSI Layout System*," Proceedings of the 21st Design Automation Conference, 1984, pp. 152-159.

[Pfi 82] Gregory F. Pfister, "*The Yorktowm Simulation Engine: Introduction*," Proceedings of the 19th Design Automation Conference, 1982, pp. 51-54.

[PrK 86] Bryan T. Preas and Patrick G. Karger, "*Automatic Placement: A Review of Current Techniques*," Proceedings of the 23rd Design Automation Conference, 1986, pp. 622-629.

[PuE 88] Douglas A. Pucknell and Kamran Eshraghian, *Basic VLSI Design – Systems and Circuits*, Prentice Hall, 1988.

[Rab 83] Guy Rabbat, ed., *Hardware And Software Concepts in VLSI*, Van Nostrand Reinhold, 1983.

[Ram 83] Vijaya Ramachandran, "*An Improved Switch-Level Simulator for MOS Circuits*," Proceedings of the 20th Design Automation Conference, 1983, pp. 293-299.

[Ram 86] Vijaya Ramachandran, "*Algorithmic Aspects of MOS VLSI Switch-Level Simulation with Race Detection*," IEEE Transactions on Computers, Vol. c-35, No. 5, May 1986, pp. 462-475.

[Riv 82] Ronald L. Rivest, "*The "PI" (Placement and Interconnect) System*," Proceedings of the 19th Design Automation Conference, 1982, pp. 475-481.

[SaB 80] S. Sahni and A. Bhatt, "*The Complexity of Design Automation Problems*," Proceedings of the 17th Design Automation Conference, 1980, pp. 402-411.

[SaH 89] Arturo Salz and Mark Horowitz, "*IRSIM: An Incremental MOS Switch-Level Simulator*," Proceedings of the 26th ACM/IEEE Design Automation Conference, 1989, pp. 173-178.

[ScO 84] Walter S. Scott and John K. Ousterhout, "*Plowing: Interactive Stretching and Compaction in Magic*," Proceedings of the 21st ACM/IEEE Design Automation Conference, 1984, pp. 166-172.

[ScO 85] Walter S. Scott and John K. Ousterhout, "*Magic's Circuit Extractor*," Proceedings of the 22nd Design Automation Conference, 1985, pp. 286-292.

[She 89] Alan T. Sherman, *VLSI Placement And Routing: The PI Project*, Springer-Verlag, 1989.

[SMB 87] Steven P. Smith, M. Ray Mercer, and Bishop Brock, "*Demand Driven Simulation: BACKSIM*," Proceedings of the 24th ACM/IEEE Design Automation Conference, 1987, pp. 181-187.

[SSS 87] John W. Smith, K. Stuart Smith and Robert J. Smith, II, "*Faster Architectural Simulation Through Parallelism*," Proceedings of the 24th ACM/IEEE Design Automation Conference, 1987, pp. 189-194.

[TaO 84] George S. Taylor and John K. Ousterhout, "*Magic's Incremental Design-Rule Checker*," Proceedings of the 21st ACM/IEEE Design Automation Conference, 1984, pp. 160-165.

[Ter 83] Christopher J. Terman, "*Simulation Tools for Digital LSI Design*," PhD thesis, Massachusetts Institute of Technology, October 1983.

[Tri 81] S. Trimberger, "*Combining Graphics and a Layout Language in a Simple Interactive System*," Proceedings of the 18th Design Automation Conference, 1981, pp. 234-239.

[Tri 85] Howard W. Trickey, "*Compiling Pascal Programs into Silicon*," Technical Report STAN-CS-85-1059, Dept. of Computer Science, Stanford University, July 1985.

[WHP 87] Laung-Terng Wang, Nathan E. Hoover, Edwin H. Porter, and John J. Zasio, "*SSIM: A Software Levelized Compiled-Code Simulator*," Proceedings of the 24th ACM/IEEE Design Automation Conference, 1987, pp. 2-8.

[WeE 85] Neil H. E. Weste and Kamran Eshraghian, *Principles of CMOS VLSI Design – A System Perspective* Addison-Wesley, 1985.

[WoL 86] D.F. Wong and C.L. Liu, "*A New Algorithm for Floorplan Design*," Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986, pp. 101-107.

[WoL 87] D.F. Wong and C.L. Liu, "*Floorplan Design for Rectangular and L-Shaped Modules*," Proceedings of the IEEE International Conference on Computer-Aided Design, 1987, pp. 520-523.

NYU COMPSCI TR-522 c.1
You, Yongtao
Toward a fully integrated
VLSI CAD system

DATE DUE	BORROWER'S NAME

DATE DUE

GAYLORD			PRINTED IN U.S.A.